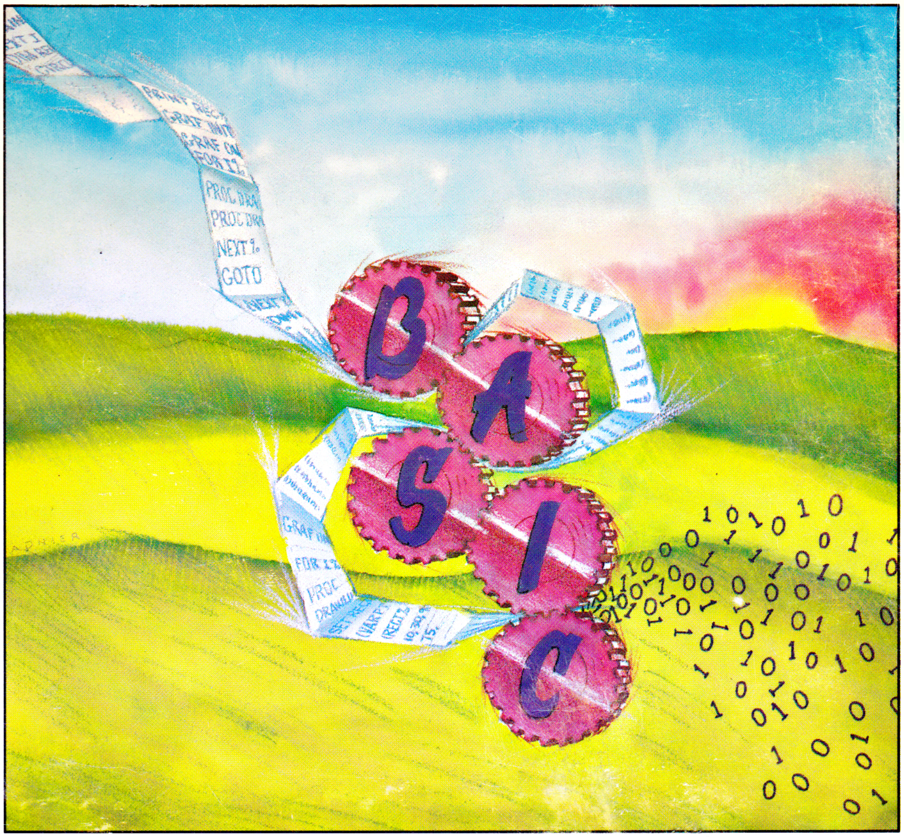


TML BASIC

for the Apple IIGS

TML SYSTEMS

83473

TML BASIC for the Apple IIGS

User Manual

Version 1.0, December 1987

TML BASIC™ LICENSE AGREEMENT

This manual and the software described in it were developed and copyrighted by TML Systems, Inc. and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except as follows:

1. You may make backup copies of the software only for your personal use providing the backup copies bear TML Systems' copyright notice.

2. You have the right to include the object code provided in the several libraries included with TML BASIC in programs you develop using this software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees providing that you include the following copyright notice *(no less prominently than your own copyright notice)* in the software and its documentation:

   "© 1987 TML Systems, Inc. Certain portions of this software are copyrighted by TML Systems, Inc."

3. You may not, in any event, distribute any of the source files or the TML BASIC application provided as part of this software.

4. You may use the software and its documentation at any number of locations or on any machine so long as there is no possibility of it being used at more than one location or on one machine at any one particular time.

CUSTOMER SUPPORT AND PRODUCT UPGRADE PLAN

Software Registration. Your registration of TML BASIC is ESSENTIAL for you to receive the full benefits of TML Systems' customer services. TML BASIC is a very large and sophisticated software package. From time to time, TML Systems will improve its product making it even more powerful and useful to you. You can take advantage of our ongoing development efforts if you have returned your registration card to us. As a registered TML BASIC user, you will receive announcements about major improvements for your software. These announcements will provide you the cost of the upgrade and ordering procedures. Only registered users will receive these upgrade notices and be eligible to purchase the upgrade.

Technical Support. We at TML Systems would like you to take the greatest advantage of your development tools as possible. If you have a technical problem we will be glad to help. Gather ALL pertinent information to recreate the problem along with your registration number, and call our Technical Support Department at (904) 636-0118 during our normal support hours. You may also write to:

TML Systems, Inc.
Technical Support Department
8837-B Goodbys Executive Drive
Jacksonville, Florida 32217

Remember, it is required that you include your registration number with all correspondence and have it available when you call TML Systems. TML Systems retains the right to deny Technical Assistance to any person unable to identify his software by registration number.

Version	Printing	Date
1.0	First Printing	December 1987

Your suggestions and input are extremely valuable in assisting us to continue providing the most complete development tools possible. If you have any comments or suggestions regarding either the TML BASIC development system software or this documentation, please send comments to:

TML Systems, Inc.
Customer Support Department
8837-B Goodbys Executive Drive
Jacksonville, Florida 32217

to Laurrie and Donna

Table of Contents

Introduction

Welcome to *TML BASIC for the Apple IIGS*. TML BASIC is a programming language designed to meet the needs of the broadest range of programmers possible for the Apple IIGS. TML BASIC is a modern, 16-bit, *compiled* implementation of the BASIC (Beginner's All-purpose Symbolic Instruction Code) language and is compatible with Apple Computer's GS BASIC, an *interpreted* implementation of the BASIC language.

TML BASIC is an extended version of the BASIC programming language and includes many new features and statements not found in more traditional implementations. For example, TML BASIC provides control structures like the DO...WHILE...UNTIL loop and block IF statements, the PRINT USING statement, user defined, multiline functions and procedures with local variables, and a mechanism for supporting separately compiled libraries of code.

Programmers familiar with AppleSoft BASIC will find TML BASIC an easier and more powerful means of developing programs to run on the Apple IIGS. Using the TML BASIC Translator (a separate product), AppleSoft BASIC programmers are capable of converting their AppleSoft BASIC programs into TML BASIC programs for increased performance and easier maintainability on the Apple IIGS.

TML BASIC is a complete programming environment which combines a compiler with a fully integrated, mouse-based, multi-window editor. Remember, TML BASIC is a compiled BASIC language. Results of a compiled language are faster and more efficient programs capable of being created and tested in TML BASIC's user-friendly environment.

TML BASIC has been designed specifically to take advantage of, and provide access to, the new features and capabilities of the Apple IIGS. TML BASIC runs in a full 16-bit native mode under ProDOS 16. Complete access to every Apple IIGS Toolbox routine, including Super HiRes graphics, Menus, Windows, etc., are provided with BASIC procedures and functions. With TML BASIC, you will be able to develop stand-alone ProDOS 16 applications capable of running independently of TML BASIC and transferable to any Apple IIGS disk.

In addition to writing programs which take advantage of the Apple IIGS Toolbox, TML BASIC allows you to write more traditional programs which use only the text screen. We call these *textbook* programs. *Textbook* programs are the type of programs you would enter directly from BASIC textbook examples and then compile them. An understanding of the IIGS Toolbox is not necessary to write a textbook program.

About this Manual

No specific knowledge of *programming* the Apple IIGS is necessary to use TML BASIC, however we do assume you are familiar with the *concept* of programming and perhaps have had some experience programming on another machine.

The TML BASIC manual is divided into four major parts. The first part of the manual is a user's guide, and its chapters discuss how to actually operate TML BASIC and write your first program. A complete TML BASIC language reference is provided in the second part, and the third part provides documentation on how to program using the Apple IIGS Toolbox. Finally, the fourth part is a collection of appendices. This manual assumes you are familiar with the Apple IIGS Finder and the machine itself.

The following paragraphs outline the information contained in each of the manual's chapters. Use these descriptions to find the information you are looking for.

Part I: TML BASIC User's Guide

TML Systems recommends you take the time to study Chapters 1 through 6 prior to beginning your actual programming work. Chapters 1 - 4 and 6 explain in detail the capabilities of the product itself, while Chapter 5 instructs you through your first program. These chapters are certain to prove useful to the programmer who has taken the time to master them. Each chapter reminds you to close all example programs opened during the chapter's discussion and to exit TML BASIC, thus assuring each chapter is treated as an independent learning session of TML BASIC's integrated environment.

Chapter 1: **Starting out with TML BASIC** shows you how to make a backup copy of TML BASIC, discusses what files are on the TML BASIC distribution disk, explains the differences between *Textbook* and *Toolbox* programming and provides a comparison of compiled versus interpreted languages.

Chapter 2: **Using TML BASIC** includes a quick tour of TML BASIC using two example programs included on the TML BASIC distribution disk. The chapter's discussion takes you from the first step of running TML BASIC to performing window manipulation commands.

Chapter 3: **Compiling and Running a Program** discusses TML BASIC's compile features while showing you how easy it is to run a TML BASIC program. Creating libraries as well as detecting and correcting errors in your program's source code is also discussed.

Chapter 4: **Advanced Program Editing** discusses some of the more powerful features of TML BASIC's integrated editor, thus enabling you to use TML BASIC more effectively in creating your own programs.

Chapter 5: **Your First Program** explains the idea of textbook programming and begins to introduce some of TML BASIC's language features by instructing you through your first TML BASIC program.

Chapter 6: **TML BASIC Menu Reference** provides a summary of TML BASIC's menus and commands. This chapter should be used as a reference to the features available within TML BASIC.

Part II: TML BASIC Language Reference

The TML BASIC Language Reference is a complete reference for the TML BASIC programming language. The first three chapters discuss various components of the language, while Chapter 10 provides a thorough discussion of each statement and function available in TML BASIC.

Chapter 7: **Language Elements** discusses the fundamental components which make up a TML BASIC program. A discussion of constants, variables, arrays and expressions is also included.

Chapter 8: **Subroutines, Procedures, Functions and Libraries** reviews the language constructs available in TML BASIC which promote modular programming for better organization of a program's code.

Chapter 9: **Files** provides a review of the techniques and operations available in TML BASIC for reading, writing and manipulating files.

Chapter 10: **Statements and Functions** is a comprehensive discussion of each statement and built-in function implemented in TML BASIC. You will find this chapter most useful during your programming efforts.

Part III: Toolbox Programming

This portion of the manual is written for experienced programmers and introduces the concept of programming the Apple IIGS Toolbox. The Toolbox is the huge collection of procedures and functions available with every Apple IIGS which implements features like the Super Hi-Res graphics screen, Menus, Windows, Dialogs, Sound, etc. Toolbox programming is not for everybody. Obviously more complicated than textbook programming, Toolbox programming provides a whole new spectrum of features you can add to your programs.

Chapter 11: **Programming the Toolbox** first reviews the contents of the Toolbox and then introduces the language features available in TML BASIC for accessing the Toolbox.

Chapter 12: **QuickDraw Graphics** is the graphics engine for the Apple IIGS which implements all of the drawing operations available for the Super Hi-Res graphics screen. Because QuickDraw is the soul of the Apple IIGS, this chapter provides a discussion and review of the principles behind this powerful graphics engine.

Chapter 13: **Creating a Desktop Application** discusses the techniques for writing programs in TML BASIC which make use of the *Desktop metaphor*. The desktop is considered the menu bar, a collection of windows, dialogs, etc.

Part IV: Appendices

This part of the manual provides a wide collection of useful information for the TML BASIC programmer. Included is a summary of the error messages generated by both the TML BASIC compiler and editor. Also included is a complete list of every Apple IIGS Toolbox routine accessible with TML BASIC.

Appendix A: **Error Messages** provides a list of every error generated by the TML BASIC editor, compiler, linker and runtime debugger. Along with each error message is a discussion of what the error message means and how it might have occurred.

Appendix B: **Metastatements** describes each of the TML BASIC compiler's metastatements. Metastatements direct the compiler to behave in a specific manner.

Appendix C: **Apple IIGS Toolbox Libraries** is a complete and exhaustive list of every Toolbox procedure and function available with TML BASIC.

Appendix D: **Comparing TML BASIC with GS BASIC** is a summary of the differences between the TML BASIC and GS BASIC languages.

Appendix E: **The ASCII Character Set**

Appendix F: **ProDOS 16 Filetypes**

Index

Apple IIGS Technical Documentation from Apple Computer, Inc.

While the Apple IIGS provides a new degree of friendliness to the user, the programmer is confronted with the burden of developing software for a much more sophisticated machine. Without the appropriate technical references, the task of programming the Apple IIGS and its Toolbox will be nearly impossible. The following paragraphs outline the technical documentation published by Apple Computer for the Apple IIGS. Each of these texts is available directly from Addison-Wesley or the Apple Programmer's and Developer's Association (APDA).

- *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals for the Apple IIGS. It describes all aspects of the Apple IIGS, including its features, general design, and Toolbox.

- *Apple IIGS Hardware Reference* and *Apple IIGS Firmware Reference* cover the hardware details of the Apple IIGS. You will not necessarily need these texts in order to develop applications for the Apple IIGS, however, reading them might provide you with a better insight as to how the machine operates.

- *Programmer's Introduction to the Apple IIGS* provides an excellent introduction to the concepts and guidelines you will need to know in order to develop quality applications which take specific advantage of the Apple IIGS. While this text uses TML Pascal for examples, you will find the information here useful for programming the Apple IIGS Toolbox with TML BASIC.

- *Apple IIGS Toolbox Reference: Volume 1 and Volume 2* is the complete and authoritative reference for the Apple IIGS's built in set of routines which are collectively known as the Toolbox. For example, the Toolbox contains the software necessary to draw graphical objects on the screen (QuickDraw) and for menus, windows, and sound. The Toolbox supports the Apple desktop user interface and simplifies development of new and powerful applications.

  If you intend to develop applications which take advantage of the Toolbox, you will find these two volumes absolutely essential. It will be nearly impossible to program the Toolbox effectively without this documentation.

- *Apple IIGS ProDOS 16 Reference* documents the operating system of the Apple IIGS. The details of the System Loader and file manipulation operations are covered in this text.

- *Human Interface Guidelines: The Apple Desktop Interface.* This book documents Apple's standards for the desktop user interface to any program that runs on an Apple IIGS or a Macintosh. If you are writing an application which is to use the desktop user interface, you should study this manual to ensure your application conforms to the standards set forth by Apple Computer.

- *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE), a full implementation of the IEEE standard for floating-point arithmetic.

In addition to these texts, Apple Computer publishes a series of *Technical Notes* for the Apple IIGS on a periodic basis. These notes discuss often asked technical questions and other mysteries about the Apple IIGS. The technical notes are available on a subscription basis from the Apple Programmer's and Developer's Association. Below is the address for the Apple Programmer's and Developer's Association.

Apple Programmer's and Developer's Association
290 SW 43rd Street
Renton, WA 98055
(206) 251-6548

Please note that in order to purchase products from APDA you must first be a member. There is a nominal annual fee required for membership into APDA.

Where to go for more Information

In addition to technical documentation from Apple Computer, you may find one or more of the following texts useful in your programming efforts.

The following three books document the Apple IIGS Toolbox. While the books do not use TML BASIC as examples, they still provide a wealth of useful information. In particular, the *Apple IIGS Technical Reference* by Michael Fischer provides exhaustive coverage of the Toolbox, but in a much more readable fashion than Apple Computer's *Apple IIGS Toolbox Reference* volumes.

- *Apple IIGS Technical Reference,* Michael Fischer, Osborne/McGraw-Hill, 1987.
- *The Apple IIGS Toolbox Revealed,* Danny Goodman, Bantam Computer Books, Prentice Hall Press, 1986.
- *Exploring the Apple IIGS,* Gary Little, Addison-Wesley, 1987.

Notational Conventions

The following notational conventions are used in this manual. Understanding these conventions will help make this manual more useful to you.

Notation	Description
Command	Bold typeface as shown in the left column and appearing within the *text* of this manual identifies commands you may enter from the keyboard or by using the mouse.
`Source Code`	The typeface shown in the left column is used to simulate the appearance of a program's source code, or both input and output, that would be printed on your screen. This notation is used for program listings as well as references made within the text of this book to a particular source code listing.
Source Code and Important words	Certain words within the *text* of this manual are italicized in order to emphasize their importance. Reference to any portion of source code (i.e. variable names) within the text of this manual also appears in italics.
HELLOWORLD.BAS	Words appearing in all upper case letters represent program (file) names contained either on the distribution disk or programs (files) you create yourself. A filename with the .BAS extension represents a program's source code. A filename without the .BAS extension represents a stand-alone application found on disk.

System Requirements

In order to use TML BASIC, you will need an Apple IIGS with at least one 3.5" 800K floppy disk drive, and a memory expansion card with at least 256K bytes of additional memory for a total of 512K RAM memory. For development of large applications, TML BASIC can be used with a hard disk and up to 8 megabytes of memory. TML BASIC supports the ImageWriter and any compatible serial printer or any compatible parallel printer with an appropriate interface card.

Acknowledgements

TML BASIC™, TML BASIC Translator ™, TML Pascal™, TML Speech Toolkit™ and TML Source Code Library™ are trademarks of TML Systems, Inc.

Apple®, Apple Computer, Inc.®, ImageWriter®, LaserWriter®, Mac®, MacWrite® and ProDOS® are registered trademarks of Apple Computer, Inc.

Apple IIGS™, GS BASIC™, Finder™, Macintosh™ and SANE™ are trademarks of Apple Computer, Inc.

Part I

TML BASIC User's Guide

Chapter 1

Starting Out with TML BASIC

Before you begin using TML BASIC, you should make a *working copy* of your distribution disk and store the original in a safe place. This chapter explains how to accomplish this task. It also describes the files on the TML BASIC distribution disk, thus enabling you to see what files are provided and which of those files you will need to use TML BASIC. A discussion of *compiled* versus *interpreted* languages is provided, as well as the use of line numbers in TML BASIC.

Before proceeding any further, you should familiarize yourself with the Apple IIGS. You should be knowledgeable in such tasks as booting your machine, using the mouse, copying files, and selecting and running applications using the Apple IIGS Finder. If you are unfamiliar with any of these operations, consult your *Apple IIGS Owner's Manual* and *Apple IIGS System Disk User's Guide* for information.

Backing up the Distribution Disk

TML BASIC is distributed on one 3.5" 800K ProDOS 16 disk and includes the Apple IIGS System Disk's files (version 3.1 or later). In the spirit of TML System's philosophy - selling software without copy protection - the distribution disk is not protected from being copied. Thus, you should make a *backup copy* of the distribution disk and store the original in a safe place. This manual refers to the backup copy of TML BASIC as the *working copy*. You should store the original TML BASIC disk and use it in only in the event of your working copy being damaged.

Although TML BASIC may be copied, your license agreement specifically states you may only do so for your own private use and only for the purpose of making a backup copy. Any other copies are not allowed and are in violation of the United States Copyright laws.

In order to make a backup copy, you will need an unused 3.5" disk and a disk copying utility. Included with the Apple IIGS System files on the TML BASIC distribution disk is the Apple IIGS Finder. The Finder includes the capability of formatting an unused disk and copying the TML BASIC distribution disk's files onto the newly formatted disk. Figure 1-1 illustrates the contents of the TML BASIC distribution disk in an open window on the Apple IIGS desktop.

Refer to your *Apple IIGS System Disk User's Guide* for information on how to use the Finder, or your *Apple IIGS Owner's Guide* for information about formatting and copying to a disk.

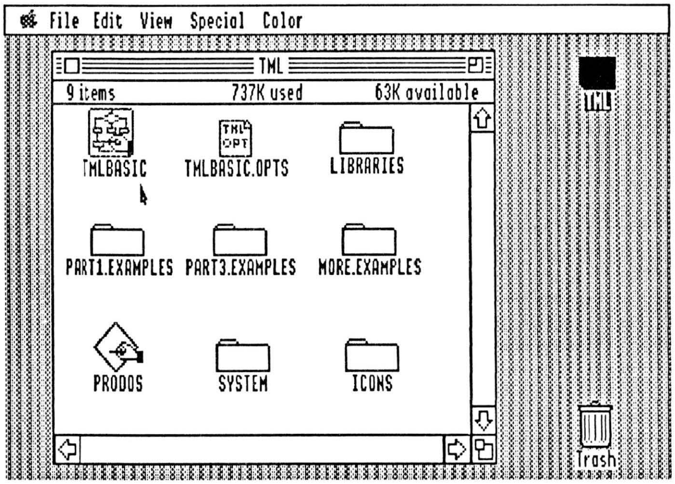


Figure 1-1
TML BASIC Distribution Disk

Files on the Distribution Disk

Table 1-1 lists the TML BASIC distribution disk's contents. The files shipped on the distribution disk can be grouped into four categories: the TML BASIC compiler, TML BASIC example programs, the Apple IIGS Toolbox libraries and the Apple IIGS system files required to boot your machine and run TML BASIC or any applications you might create.

Remember, not all the files included on the distribution disk are required to *run* TML BASIC. In the following table, the files *required* to run TML BASIC are listed in **boldface**, while the others are listed in normal typeface. The files listed below are for the version 1.0 TML BASIC distribution disk. Subsequent releases of this product may include different files.

Table 1-1
TML BASIC Distribution Disk Contents

/TML/	The name of the TML BASIC distribution disk.
TMLBASIC	The TML BASIC compiler.

TMLBASIC.OPTS	This file saves various options for using TML BASIC such as tab width, printer port, etc.
LIBRARIES/	This folder (subdirectory) contains all of the library interface file for the Apple IIGS Toolbox. These files are described in detail in Chapter 11 and Appendix C.
PART1.EXAMPLES/	This folder (subdirectory) contains the source code to the example BASIC programs used in Part I of the TML BASIC User Manual.
PART3.EXAMPLES/	This folder (subdirectory) contains the source code to the example BASIC programs used in Part III of the TML BASIC User Manual.
MORE.EXAMPLES/	This folder (subdirectory) contains the source code to the several additional example BASIC programs which demonstrate many of the capabilities of TML BASIC.
PRODOS	The ProDOS file that is used to begin the booting process of your Apple IIGS.
SYSTEM/	A folder (subdirectory) containing the ProDOS 16 and Apple IIGS system files necessary to use the Apple IIGS. This folder contains only a subset of the files found on the complete Apple IIGS System Disk necessary for TML BASIC.
P16	The ProDOS 16 operating system.
START	A program which determines whether or not to run the Program Launcher or the Apple IIGS Finder.
LAUNCHER	The Apple IIGS Program Launcher.
FINDER	The Apple IIGS Finder.
SYSTEM.SETUP/	A folder which contains any necessary boot time initialization files for the Apple IIGS.
TOOLS/	A folder which contains all of the RAM based Apple IIGS Toolbox toolsets.
DESK.ACCS/	A folder which contains classic and new desk accessories. This folder contains only the TML Clock new desk accessory.

DRIVERS/	A folder which contains the various printer and modem drivers.
FONTS/	A folder which contains Apple IIGS font files. These files are used by the Font Manager. This folder is empty on the TML BASIC disk.
ICONS/	A folder which contains *icon definition files*. These files are used by the Finder to display applications and documents with their icons.

Textbook versus Toolbox Programming

The introduction of this book mentioned two different types of programs capable of being written in TML BASIC. The first type is referred to as a *textbook* program and represents the kind of program typically found in most BASIC programming textbooks – traditional programs that use the computer's text screen. The second type of program makes use of the special features and capabilities of the Apple IIGS Toolbox.

Chapter 5 of this manual discusses *textbook* programming techniques and requires all of the **boldface** files and directories listed in Table 1-1 be present on your working copy of the TML BASIC distribution disk. In addition, you will need the AVERAGES.BAS example program found in the PART1.EXAMPLES folder.

Part III of this manual introduces the concept of programming the Apple IIGS Toolbox and describes the contents of the IIGS Toolbox. In addition to the **boldface** filenames appearing in Table 1-1, programs designed to use the IIGS Toolbox will require the files found in the folder LIBRARIES. You may also wish to have the files in the folder PART3.EXAMPLES.

Setting Up

The following three sections describe how you might set up a working environment for using TML BASIC with a single 3.5" 800K disk drive system, a dual disk drive system using either two 3.5" 800K or one 3.5" 800K and one 5.25" disk drives, or a hard disk.

NOTE

As noted earlier, TML BASIC is shipped with the contents of the Apple IIGS System Disk **version 3.1**, or later, and includes the Apple IIGS Finder. The Finder requires a minimum of 512K RAM. On startup, System 3.1 identifies the amount of memory available. If 512K or greater memory is available, the Finder is displayed. If only 256K RAM is available, neither the Finder nor TML BASIC will run. This manual assumes your system includes at least 512K RAM and one 3.5" disk drive.

Single Disk Drive System

Users with a single 3.5" 800K floppy disk will find that TML BASIC can be used exactly as it is shipped on the distribution disk without having to sacrifice any functionality or performance. You should create a working single disk system by making a copy of the distribution disk as described in the section *Backing up the Distribution Disk* in the beginning of this chapter.

The only restriction imposed by the single disk system is the size of the programs you develop will be restricted to available disk space to store them. On your *working copy* of TML BASIC, feel free to delete the various folders containing example programs. The PART1.EXAMPLES folder includes the example programs used in the first part of this manual which you should keep on your disk if you intend to follow the discussions in the next four chapters.

You will still have access to all example programs you choose to delete by copying them from the original distribution disk as needed. *Never* delete any of the files from the original distribution disk.

Two Disk Drive System

If you have a second disk drive, either another 3.5" 800K disk drive or 5.25" disk drive (formatted for ProDOS 16 of course), then you can take advantage of this extra storage capacity for developing larger programs. You may find it easier to keep all of the example programs, as well as any new programs you create, on a separate disk and access them using your second disk drive. The LIBRARIES folder (see Table 1-1) should be kept on the TML BASIC disk, thus allowing the folder's files to be shared by all programs.

Hard Disk Drive System

While a hard disk is not required to use TML BASIC, you will enjoy the luxury of faster disk access and an extensive amount of disk storage space available for creating large programs. To use any ProDOS 16 formatted hard disk drive with TML BASIC simply copy the necessary TML BASIC files onto your hard disk as outlined in Table 1-1. If your hard disk contains Apple IIGS System files prior to version 3.1, TML BASIC will not work. In this case, you should copy the System files from the distribution disk onto your hard disk.

A Note on RAM Disks

Traditionally, Apple II users have found the use of RAM disks advantageous, and have done so without "stealing" available memory from an application, due to the Apple II's restriction of permitting only 128K of memory or less to a single application.

TML BASIC and the Apple IIGS are different however. TML BASIC is a memory resident application, so there is no advantage in storing TML BASIC on a RAM disk. Further, TML BASIC maintains in memory, an entire copy of the file(s) it is editing; inlcuding library files, compiled code, etc., and uses the Apple IIGS Memory Manager to keep track of available memory. Thus, any RAM space you might allocate for a RAM disk would only decrease the amount of memory TML BASIC has available to it for editing and compiling.

Compiled versus Interpreted Languages

TML BASIC is a *compiled* language. In this regard, as well as others, it differs from *interpreted* languages such as Apple Computer's GS BASIC. A programming language is characterized by its collection of statements, expressions and other components generally known as the syntax, or structure, of the language. While programs written in a computer language are generally understandable to the human reader, they are totally incomprehensible to the computer, or in the case of the Apple IIGS, the 65816 microprocessor.

Before a statement written in a computer language can be executed by the computer, it must first be translated into code understood by the computer - *machine language.* Machine language consists of long lists of binary numbers (0's and 1's) that are understood by the computer as a series of off and on states representing operations the computer is capable of performing. Of course, a long string of 0's and 1's is not easily understood or readily comprehended by humans.

A major part of any computer language system is its means of translating programs into machine language. In an interpreted language, the translation is done while the program is being executed, sometimes denoted as "on the fly". If a statement in the program is executed 100 times, the translation is also done 100 times. Interpreted languages run slower than compiled languages because of the need for translation to occur during the running of the program.

With a compiled language, however, the translation of programs into machine language is handled prior to running the program. Thus, each line in the program is translated only once - during the compilation process. In addition, the compilation process discloses all of the syntax errors before the program is executed. Of course, it can't find errors in the program's logic such as *infinite loops*. Compiled programs run significantly faster than interpreted programs and they can also run independently of any language processor. That is, compiled TML BASIC programs can run by themselves under ProDOS 16 without TML BASIC on the disk.

Additionally, TML BASIC implements an integrated environment as a memory-resident application that compiles programs with the same interactiveness as an interpreter. This means that TML BASIC integrates its editor, compiler and the running program into memory at the same time, thus eliminating the need to read and write disk files which cause conventional compilers to be so much slower than an interpreter in translating a BASIC program.

Line Numbers in TML BASIC

Historically, BASIC implementations have required the use of line numbers in programs, however, TML BASIC does not require line numbers. In fact, TML BASIC does not even allow the use of line numbers.

Interpretive BASIC language implementations *require* line numbers in their program source codes so that the interpreter can locate statements and functions at execution time that are not in sequential order. Line numbers are also used as a fundamental component in an interpreter's editing process.

TML BASIC has no need for line numbers as it uses *alphanumeric labels* to locate statements and functions in a program's source code. For example, rather than entering GOTO 1000, in TML BASIC you would enter GOTO *SetupProcess*, where *SetupProcess* is a *alphanumeric label* used to identify the *SetupProcess* routine. Use of alphanumeric labels is illustrated in most of the example programs discussed in Part III of this manual, as well as the example programs contained on the TML BASIC distribution disk.

Chapter 2
Using TML BASIC

In Chapter 1, you created a working copy of the TML BASIC distribution disk. Now it's time to run TML BASIC and begin learning about some of the program's capabilities. Before continuing, be certain you are using the *working copy* of TML BASIC and you have stored the original TML BASIC *distribution disk* in a safe place.

In this chapter, you will learn about the steps necessary to invoke TML BASIC from the Finder and how to perform file manipulation commands. Rather than creating a *new* BASIC program to demonstrate these tasks, this chapter's discussion will use *example* programs included within the PART1.EXAMPLES folder on your TML BASIC working disk.

Running TML BASIC

Insert your working copy of TML BASIC into the 3.5" floppy disk drive and turn (*boot*) the machine on. After the Apple IIGS completes its booting process you will be presented with the Apple IIGS Finder's desktop. Figure 2-1 illustrates the desktop's appearance after booting your Apple IIGS.

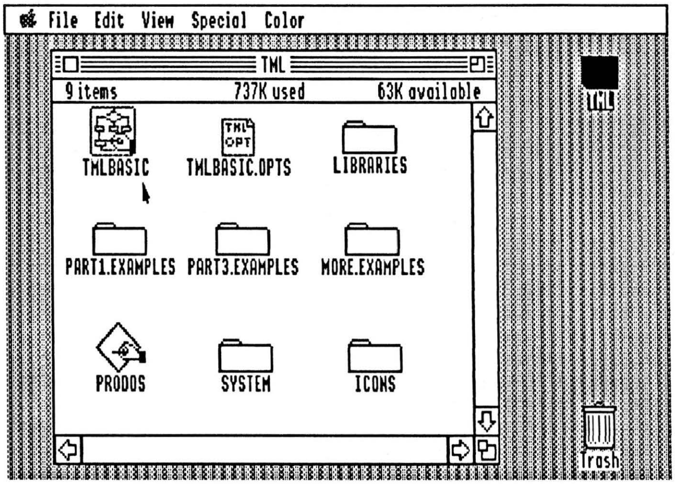


Figure 2-1
Apple IIGS Desktop

The desktop will appear as shown in Figure 2-1 only if your working copy of TML BASIC contains all of the files contained on the original TML BASIC distribution disk, and if you *booted* your computer using that disk. The desktop's appearance will differ if you are using some other hardware configuration (i.e. hard disk) or arrangement of files on your working copy of TML BASIC.

Invoke TML BASIC by clicking the mouse once on the TML BASIC icon shown on the desktop, pull-down the Finder's **File** menu and then select the **Open** command (double-clicking the mouse over the TML BASIC icon accomplishes the same result as selecting the **Open** command).

Opening the TML BASIC file results in a *splash screen* displaying the TML BASIC logo. This splash screen tells you TML BASIC is loading into the Apple IIGS's internal memory. Be patient, as the Apple IIGS requires a few moments before it completes the loading process. Figure 2-2 illustrates TML BASIC's desktop (Main Menu) after the program has been successfully loaded into memory.

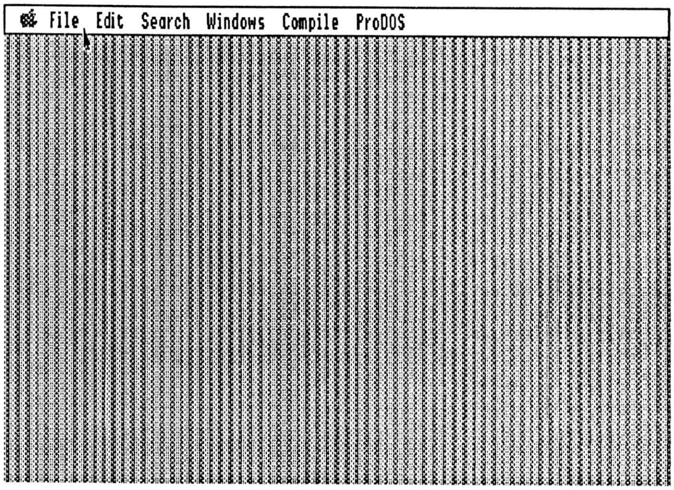


Figure 2-2
TML BASIC's Main Menu

Examining the Integrated Environment

TML BASIC has been designed to take full advantage of the Apple IIGS desktop interface using the mouse, pull-down menus, windows, etc. This user-friendly environment makes programming easy, as it integrates TML BASIC's editor and compiler into the same working environment.

The seven menus implemented in TML BASIC are designed to logically organize the several commands available to you in TML BASIC. Using the mouse, pull-down each menu to discover just how easy it is to use TML BASIC. Within each menu, you will find the various TML BASIC commands. Most commands found in the menus may be invoked by typing its corresponding *command-key equivalent* rather than pulling down its menu and selecting the command with the cursor. Command-key equivalents are displayed next to their command names in each of the pull-down menus. Chapter 6 provides a review of each TML BASIC menu, its corresponding commands and command-key equivalents.

Editing windows are the tools TML BASIC provides you for entering and modifying program source code. TML BASIC allows you to have up to four different program source codes open at one time. Each program is placed in a different editing window and is independent of any other open windows. Only one editing window can be *active* at a time. TML BASIC identifies the active window as the window which is *topmost*. All commands issued by the user are performed on the source code contained in the *active window*. Figure 2-3 illustrates the various components of a typical TML BASIC editing window.

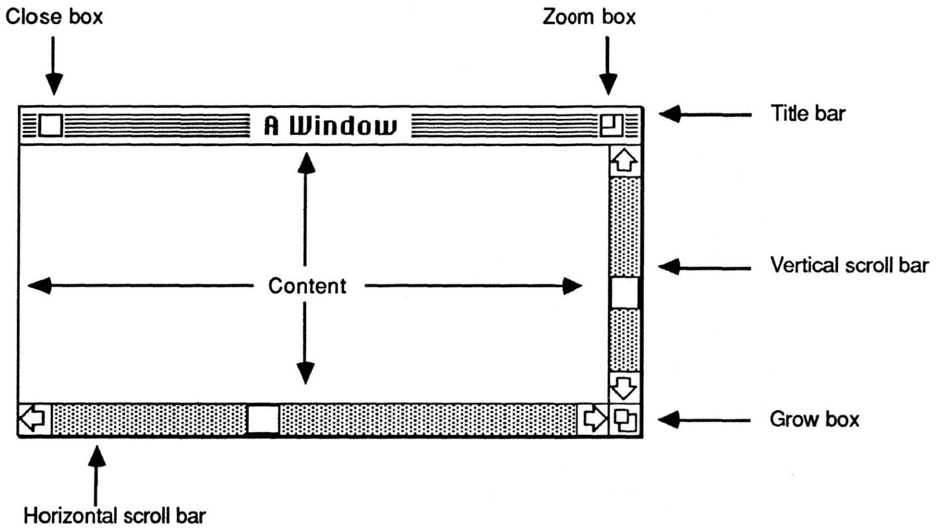


Figure 2-3
Editing Window Features

TML BASIC also implements *dialog boxes* as a means of communicating with the user. The different dialog boxes used in TML BASIC will be discussed individually as appropriate throughout the chapter discussions ahead. However, a brief mention of what a dialog box is and how it works is discussed in the following paragraph.

A dialog box is a window whose appearance is different than the editing windows used in TML BASIC. Namely, it does not have a title across the top of the window, nor does it have scroll bars and it can not be moved around on the screen. A dialog box is used in TML BASIC to provide the user with requested information, or to ask the user for required information before continuing. Dialog boxes usually inlcude OK/Cancel *buttons* or Yes/No buttons. These buttons allow you to communicate with TML BASIC to signify when you are finished with the dialog box.

Opening a Program

We will begin our tour of TML BASIC by opening a few example applications on the TML BASIC disk. TML BASIC is capable of opening up to four separate editing windows at one time, each independent of the other and containing a different program. This feature provides you the flexibility of studying the source code of two or more programs at the same time, or even copying code from one program to use in another. Chapter 4 discusses the different techniques for copying source code from one program to use in another.

We will start by opening the TML BASIC Open File Dialog Box, which lists the files available on the TML BASIC disk. You can accomplish this by either dragging the cursor down the **File** menu and then releasing the cursor on the **Open** command, or by typing the **Open** command-key equivalent ⌘O Regardless of the method you use, the result on your screen should appear similar to Figure 2-4.

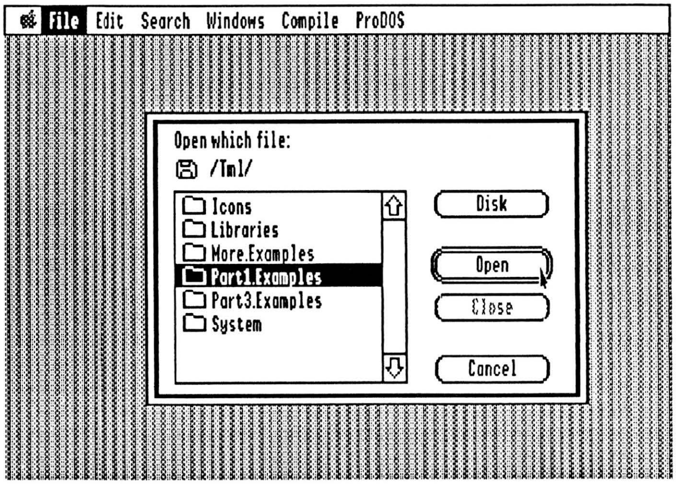


Figure 2-4
Open File Dialog Box

The Open File Dialog Box displays the files and folders contained on the TML BASIC disk. Now, click the mouse on the PART1.EXAMPLES folder once and then on the **Open** button. The files contained in the PART1.EXAMPLES folder now appear in the Open File Dialog Box as illustrated in Figure 2-5. Find the HELLOWORLD.BAS file and click the cursor over its name and then on the **Open** button to open the file.

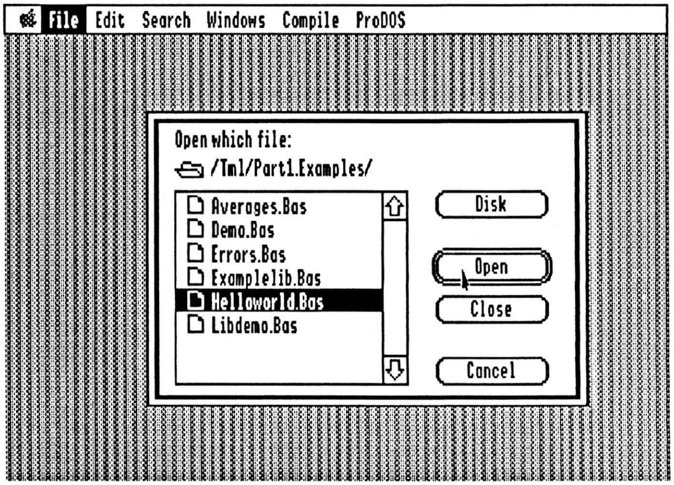


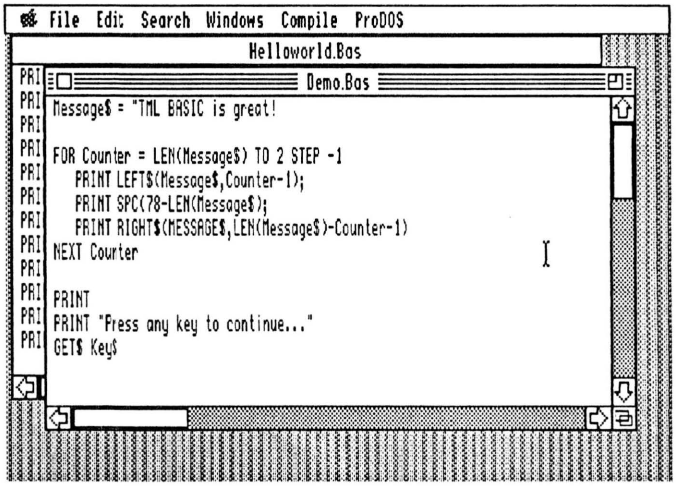
Figure 2-5
PART1.EXAMPLES Folder

If your system is operating from either a hard disk or two disk drive system, click the mouse on the Open File Dialog Box's *Drive* button until you find the PART1.EXAMPLES folder containing the TML BASIC example programs.

Once you have selected the HELLOWORLD.BAS example program, its source code is read from the disk and placed in a newly created editing window titled HELLOWORLD.BAS. Now, open a second example program, DEMO.BAS from the PART1.EXAMPLES folder, using the same technique described above.

Organizing the Editing Environment

Figure 2-6 illustrates both the HELLOWORLD.BAS and DEMO.BAS windows open and overlapping each other. TML BASIC provides you the ability of arranging your *open windows* at any location on the screen so that you can see the source code in both windows. Organizing windows can be accomplished either by dragging each window with the mouse or by invoking one of TML BASIC's window commands.



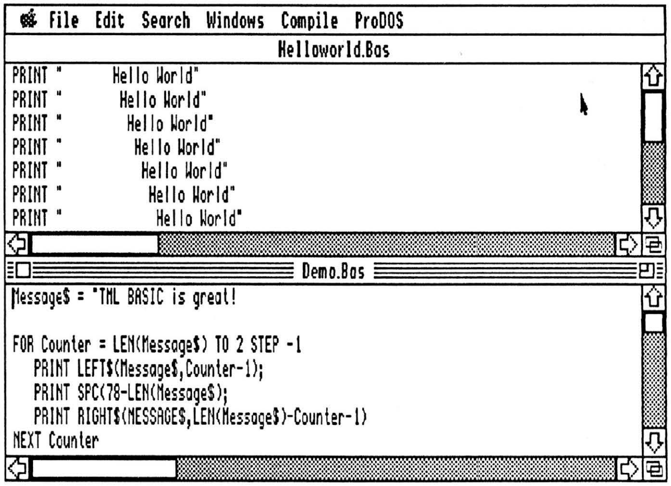
```
Message$ = "TML BASIC is great!

FOR Counter = LEN(Message$) TO 2 STEP -1
   PRINT LEFT$(Message$,Counter-1);
   PRINT SPC(78-LEN(Message$);
   PRINT RIGHT$(MESSAGE$,LEN(Message$)-Counter-1)
NEXT Counter

PRINT
PRINT "Press any key to continue..."
GET$ Key$
```

Figure 2-6
Overlapping (Stacked) Windows

Click once on the HELLOWORLD.BAS window's *title bar* using the mouse, and *drag* the window anywhere on the screen. You should notice when you first click the mouse on the window it immediately makes HELLOWORLD.BAS the *active window,* if the window is not already active, and places it in front of the DEMO.BAS window. To change the size of an editing window you use the *grow box.* The grow box is the small box in the bottom right corner of the editing window which has two small rectangles in it. Now change the size of the window by clicking the mouse once in the grow box of the HELLOWORLD.BAS window and dragging the mouse anywhere on the screen.

Pulling down the **Window** menu displays TML BASIC's window commands. After mastering the mouse techniques to change each open window's location, pull-down the **Window** menu and select **Stack Windows**. The result should arrange the windows similar to when they were originally opened as shown in Figure 2-6.

Now, select the **Next Window** command and notice how the editor places the HELLOWORLD.BAS window behind the DEMO.BAS window and makes the DEMO.BAS window the active window. Finally, selecting **Tile Windows** results in the two open windows appearing in a tile format as shown in Figure 2-7. When two or more windows are opened at one time, placing the windows in a tiled position allows you to see each program's source code at the same time.



```
PRINT "     Hello World"
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello World"
```

```
Message$ = "TML BASIC is great!

FOR Counter = LEN(Message$) TO 2 STEP -1
   PRINT LEFT$(Message$,Counter-1);
   PRINT SPC(78-LEN(Message$);
   PRINT RIGHT$(MESSAGE$,LEN(Message$)-Counter-1)
NEXT Counter
```

Figure 2-7
Tiled Windows

Program Integrity

Thus far you have learned how to open editing windows (each containing program source code) as well as the various techniques for reorganizing windows on TML BASIC's desktop. TML BASIC makes opening and rearranging windows easy with its three window commands. TML BASIC also provides you a safe means of maintaining the integrity of your program's source code in the event an unintentional change to the program's source code has been made.

The example programs opened in this chapter will be of use again later in this manual's discussions. Therefore, it is important not to alter their original content. Choosing the **Revert** command from the **File** menu directs TML BASIC to ignore any changes inadvertently made to a program's source code since it was last saved. When selecting this command, TML BASIC re-reads the last version of the program's source code from disk and places it into the editing window, thus ignoring all changes that have been made to the source code.

The **Revert** command should be used anytime an unintentional change has occurred in a program's source code. Remember, every change made to the program's source code since the *last save* will be *lost* as a result of issuing the **Revert** command. TML BASIC will display a dialog box asking if you are certain about discarding the changes made to the program prior to reverting your changes.

When you close an editing window or quit from TML BASIC, and you have not yet saved the changes made to a program, TML BASIC will ask if you would like to save the changes made. At this point you have one last chance to decide if you want to lose or keep your editing changes, or cancel the **Close** command altogether.

Exiting TML BASIC

In this chapter, we opened two example programs contained on the working disk, discussed rearranging windows in the editing environment and defined a means to avoid having unintentional changes saved in a program's source code.

Before leaving this chapter, you should close all open windows, exit TML BASIC and turn the computer off just as you would clean-up your desk before leaving for the day.

To close the open windows, select one window at a time by clicking the mouse anywhere on an open window and then choose **Close** from the **File** menu (clicking a window's close box accomplishes the same result). If changes were made to either program's source code, the Close File Dialog Box will appear asking if you would like to save those changes. Be sure to click the *No button,* thus ensuring the original program source code's integrity. After closing both files, select **Quit** from the **File** menu to exit TML BASIC and return to the Finder's desktop.

Chapter 3
Compiling and Running a Program

In Chapter 2, several of TML BASIC's file and window commands were discussed. In this chapter, we will explore the three different compile options available in TML BASIC allowing you to compile and run programs. To do this, we will re-open the same two examples discussed in Chapter 2.

Looking at Examples

Begin by booting your Apple IIGS with your TML BASIC disk and then run TML BASIC by double-clicking on the TML BASIC icon.

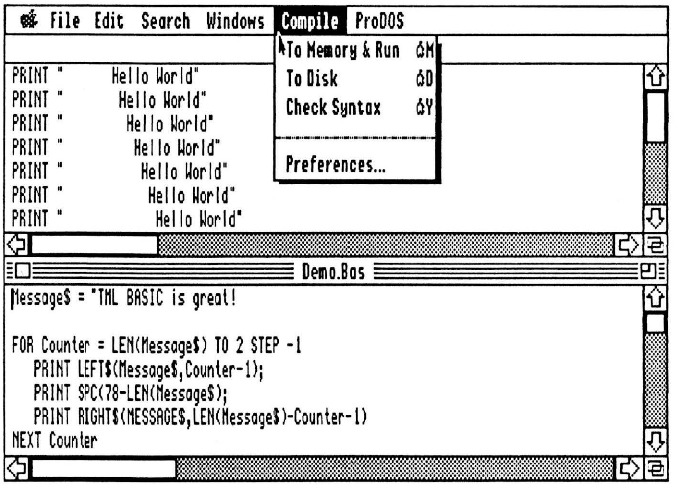
The HELLOWORLD.BAS program used in Chapter 2 is a simple input/output (I/O) program. The program is written to demonstrate how TML BASIC writes the line of text "Hello World" as output to the screen and then recognizes the carriage return key from the keyboard as input to the program. The DEMO.BAS example program uses the same I/O capabilities as HELLOWORLD.BAS but tests various string functions of TML BASIC.

Before we begin, let's re-open the HELLOWORLD.BAS and DEMO.BAS example programs. Recall that to open these programs you select the **Open** command from the **File** menu, then open the PART1.EXAMPLES folder and select the appropriate filenames from the Open File Dialog Box.

After opening both programs, the TML BASIC editing environment will consist of two open windows containing each program's source code. Figure 2-6, in Chapter 2, illustrates what your screen should look like as a result of opening both programs. You may wish to reorganize the two open windows so that both program source codes are visible - issue the **Tile Windows** command from the **Windows** menu.

Compiling Alternatives

TML BASIC offers the programmer three different options for compiling programs. The compile commands are found in the **Compile** menu. You can see each of the commands by pressing and holding the mouse button down over the **Compile** menu. Figure 3-1 shows the two open editing windows in a tiled position with the Compile menu pulled down.



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello World"
PRINT "            Hello World"
```

```
Message$ = "TML BASIC is great!

FOR Counter = LEN(Message$) TO 2 STEP -1
   PRINT LEFT$(Message$,Counter-1);
   PRINT SPC(78-LEN(Message$);
   PRINT RIGHT$(MESSAGE$,LEN(Message$)-Counter-1)
NEXT Counter
```

Figure 3-1
Compile Menu

The first command in this menu is likely to be the one you use most often. The **To Memory & Run** command invokes TML BASIC to compile the source code in the *active* editing window (the topmost window), and then, upon successful completion, executes the program directly within the Apple IIGS's internal memory.

The **To Disk** command is used to invoke TML BASIC to compile a program and create a stand-alone ProDOS 16 application file on disk. You will use this command when you have a complete running program free of errors and you wish to execute the program directly from the Finder.

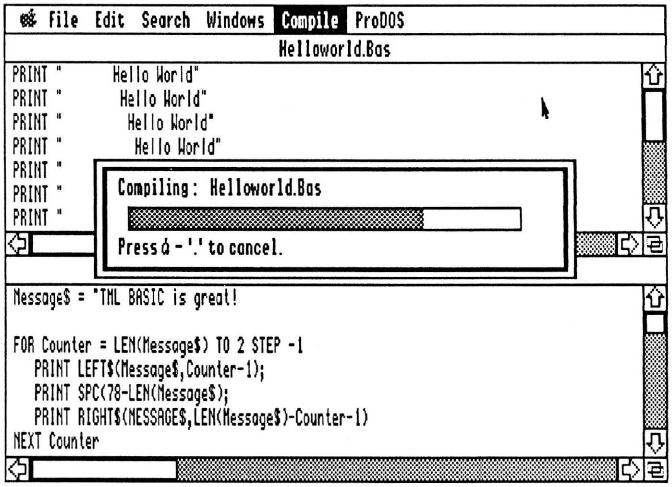
Finally, the **Check Syntax** command allows you to quickly verify the syntax of a TML BASIC program. This option does not run the selected program nor does it create a disk file. This is the fastest compile option available in TML BASIC.

When a compile option is invoked by selecting any of the three compile commands, TML BASIC displays the Compile Progress Dialog Box. This dialog is used to display the compiler's progress during compilation. When the Compile Progress Dialog Box's *indicator bar* reaches the right side of the display, the compile process has been completed.

Testing a Program's Source Code

The **Check Syntax** command is the fastest of the three compilation techniques since it does not cause any code to be generated. Instead, this command instructs TML BASIC to verify the active program was written using valid BASIC key words, statements and functions. It cannot, however, check a program for correct logic. For example, an infinite loop in a program's source code will go undetected by the **Check Syntax** command.

Click the mouse once on the HELLOWORLD.BAS window making it the active window. Pull-down the **Compile** menu and select the **Check Syntax** command. The Compile Progress Dialog Box is immediately displayed indicating the compiler's progress as it checks the syntax of the source code - Figure 3-2.



```
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "
PRINT "
PRINT "
```

```
Compiling: Helloworld.Bas

Press ơ-'.' to cancel.
```

```
Message$ = "TML BASIC is great!

FOR Counter = LEN(Message$) TO 2 STEP -1
   PRINT LEFT$(Message$,Counter-1);
   PRINT SPC(78-LEN(Message$);
   PRINT RIGHT$(MESSAGE$,LEN(Message$)-Counter-1)
NEXT Counter
```

Figure 3-2
Compile Progress Dialog Box

When the indicator bar inside the Compile Progress Dialog Box reaches the right side of the display, the compile is complete. As you will see, TML BASIC takes only a brief moment to compile the HELLOWORLD.BAS program. The reason for this, of course, is that TML BASIC is a fast compiler. In addition, the program is quite small and the **Check Syntax** command is the fastest of the TML BASIC's three compile options.

A result of no errors found does not necessarily mean a program is completely free of all possible errors. However, using the **Check Syntax** command will ensure the program does not contain any syntax errors.

It is important you use the **Check Syntax** command when you are uncertain whether your program will run correctly. Since this command does not run the program after compiling it, you can avoid situations where your program contains *logic* errors which might cause the computer to crash.

If an error is detected in the source code of a program, TML BASIC will stop the compilation process, return to the TML BASIC editor, highlight the exact location of the discovered error and then display a descriptive error message. Errors are discussed later in this chapter in the section "Detecting Program Errors".

Running a Program

Once you have determined your program does not contain any syntax errors by issuing the **Check Syntax** command, the program can then be *run*. To do this, select the **To Memory & Run** command from the **Compile** menu. Upon selecting this command, TML BASIC again displays the Compile Progress Dialog Box. This time the compiler generates code for the program. If the program does not contain syntax errors the compiled program is immediately run.

To run a compiled program, the TML BASIC environment temporarily shuts down by hiding its menus, windows, etc. and then transfers control to the compiled program. The compiled program is now in complete control of the computer as it executes. When the program has completed execution, the TML BASIC environment restores its menus and windows allowing you to continue programming.

Because it is possible the compiled program may contain logic errors causing the machine to crash, TML BASIC provides a safety feature called *Auto Save*. If this option is turned on, TML BASIC automatically saves any changes you have made to the program's source code prior to compiling. This feature ensures you will not lose your source code changes in the event of a catastrophic error during your program's execution. The Auto Save option is discussed in more detail in Chapter 6 under the "Preferences..." section.

To compile the HELLOWORLD.BAS program, first, be certain the program is in the active window (topmost window). If it is not, make it the active window by clicking the mouse once anywhere in its window. Now select the **To Memory & Run** compile command. The HELLOWORLD.BAS program uses the text screen to display the message "Hello World" at several locations on the screen. The program then waits for the Return key to be pressed. After the Return key is pressed, program execution terminates and control is returned to the TML BASIC environment with the windows restored exactly as you left them.

Creating a Stand-Alone Application

As seen above, the ompile to memory feature of TML BASIC is extremely fast and interactive. However, there exist one small problem – you must launch TML BASIC every time you want to run a TML BASIC program. Thus, the third compilation technique available in TML BASIC – **To Disk**. This compile option allows you to create stand-alone ProDOS 16 applications that can be run from the Apple IIGS Finder by double-clicking on its icon just as you did the TML BASIC icon to invoke TML BASIC. You can even copy the compiled application to another disk and run it from there because TML BASIC is no longer required after the program is compiled to disk.

Let's compile the HELLOWORLD.BAS program to disk. Make the open window containing the HELLOWORLD.BAS program the active window (remember the compile commands only work on the active window). Select the **To Disk** command from the **Compile** menu to compile the HELLOWORLD.BAS program and create a stand-alone application on disk. You will notice the compilation process takes significantly longer to complete this time.

The reason for this additional amount of time results from the compiled program being written to disk. The name of the resulting application file on disk is HELLOWORLD, and it is located in the same folder as the HELLOWORLD.BAS source code file.

IMPORTANT

Following are the three rules used by TML BASIC to determine a compiled application's filename when issuing the **To Disk** compile command.

(1) If the name of a source code file ends with the suffix .BAS then the application file is assigned the same name as the source code file less the .BAS suffix. The application file is placed in the same directory as the source code file.

(2) If the name of a source code file does not end with the suffix .BAS then the name of the application file is the name of the source code file with the letters "APP" added to the end of the name. If the source code filename is greater than 12 characters, TML BASIC uses only the first 12 characters of the source filename. The application file is placed in the same directory as the source code file.

(3) If the source code is in a new "Untitled" window, that is, there exist no disk file containing the new program's source code, then the name of the application file becomes UNTITLEDAPP. The application file is placed in the folder currently open when the Open File Dialog Box is displayed.

Compiling Libraries

In addition to compiling *programs*, TML BASIC is capable of compiling *libraries*. A library contains BASIC program statements, but is not capable of being *run* (executed) like a program. Instead, a library is compiled separately from a program and then used in one or more different BASIC programs. Libraries allow you to split a program up into smaller, more manageable pieces of code. Chapters 7 and 8 describe libraries in greater detail.

A library looks much like a program except it begins with the statement **`DEF LIBRARY`** and ends with the statement **`END LIBRARY`**. The file EXAMPLELIB.BAS in the PART1.EXAMPLES folder is an example of a TML BASIC library. Additionally, the file LIBDEMO.BAS within the PART1.EXAMPLES folder shows how the EXAMPLELIB.BAS library's source code is used in a program. Using TML BASIC, you should open these two files to see how the library mechanism is used in TML BASIC.

The EXAMPLELIB.BAS file is a library containing a procedure declaration that prints the message "Hello World" just like the HELLOWORLD.BAS program. The LIBDEMO.BAS file is a program which has only four lines of code. However, when this program is run, it generates the same output as the HELLOWORLD.BAS program because it calls the procedure in the EXAMPLELIB.BAS library.

Because a library is not capable of being *run*, the TML BASIC compiler acts differently when selecting the various compiler commands in the **Compile** menu. As mentioned above, when a library is compiled, it does not create a program that can be run. However, TML BASIC does save the library's compiled code so that other programs can use it. Thus, when selecting the **To Memory & Run** command, TML BASIC compiles the library but then returns control to the editor instead of transfering control to the compiled code as it would do for a program. Note that TML BASIC does save the compiled code in memory so that it can later be used by a program. To experiment, compile the EXAMPLELIB.BAS file by selecting the **To Memory & Run** command. Then compile the LIBDEMO.BAS program using **To Memory & Run** as well. Because the LIBDEMO.BAS file is a program, it is run immediately after successfull compilation

Libraries can also be compiled using the **To Disk** command. When a library is compiled to disk, it does not create a ProDOS 16 application, but rather, a TML BASIC .LIB file. The .LIB file contains the library's compiled source code. When a program needs to use a library that has not been compiled to memory using the **To Memory & Run** command, TML BASIC searches for the compiled code on disk in a .LIB file. Try compiling the EXAMPLELIB.BAS file using the **To Disk** command and then look on the disk using the Apple IIGS Finder for its .LIB file.

IMPORTANT

Following are the rules used by TML BASIC to determine a compiled library's filename when issuing the **To Disk** compile command.

(1) TML BASIC uses the name of the library as specified in the **DEF LIBRARY** statement as the base name of the compiled library file. TML BASIC then adds the suffix .LIB to the end of the library name to create the complete filename. The library file is placed in the same directory as the source code file.

The source code filename has no effect on the name of the compiled library file. However, to avoid confusion, it is recommended that the source code filename be the same as the library name with the .BAS suffix.

(2) If the library name is greater than 12 characters, TML BASIC uses only the first 12 characters of the library name. The .LIB suffix is then added, and the file is created in the same directory as the source code file.

The **Check Syntax** command behaves exactly the same for both programs and libraries. That is, TML BASIC only verfies the library's source code contains legal BASIC statements.

Detecting Errors

So far in this chapter we have discussed how to compile programs using TML BASIC. However, our discussion has been limited to programs known to be correct, that is, they do not contain any errors. In this section, we will discover how TML BASIC deals with errors.

First, let's consider the components of the TML BASIC environment. TML BASIC is an integrated development tool made up of three separate pieces – the editor, the compiler and the linker. These different pieces work so closely together the user really only perceives them as one in the same. However, knowing how these pieces work together will help you understand the error messages TML BASIC reports to you.

The editor of course, is where you spend most of your time. It is responsible for the editing windows and most of the commands available in each TML BASIC menu. The compiler is invoked whenever you select any of the three compile commands. The compiler is responsible for checking if syntax errors exist in your program and then generating code for the program. Finally, the linker component of TML BASIC is only invoked when you have chosen to compile a program to memory or to disk. The linker is responsible for combining the compiled code with other pieces of code your program needs (i.e. libraries). It is also responsible for allocating the internal memory a program requires in order to run within the Apple IIGS's memory, and for writing a compiled program to disk.

The editor only reports errors related to the editing environment. It will report an error when you ask it to find a string in a program that does not exist, when there is not enough memory to read another program into memory, and other operations related to the editing environment. The compiler only reports errors related to illegal BASIC source code. If you misspell a reserved word or forget to put a comma where one was expected, the compiler reports an error. Finally, the linker reports errors when an attempt to create a final program fails. This might happen if you compile a program to disk and the disk is locked or there is not enough room to fit the compiled program on disk.

When any component of the TML BASIC environment detects an error it first takes whatever actions necessary to recover without causing any loss of data and then displays the Error Dialog Box with a descriptive error message. In addition to the error message, an *icon* on the left side of the dialog box is also displayed. This icon is used to indicate which component of TML BASIC detected the error. The icon can usually help you better understand the error message. In addition, if the error is related to a particular part of your program's source code, the editor displays that portion of source code in the editing window and highlights the exact location of the error. Highlighting usually occurs for detected compiler errors.

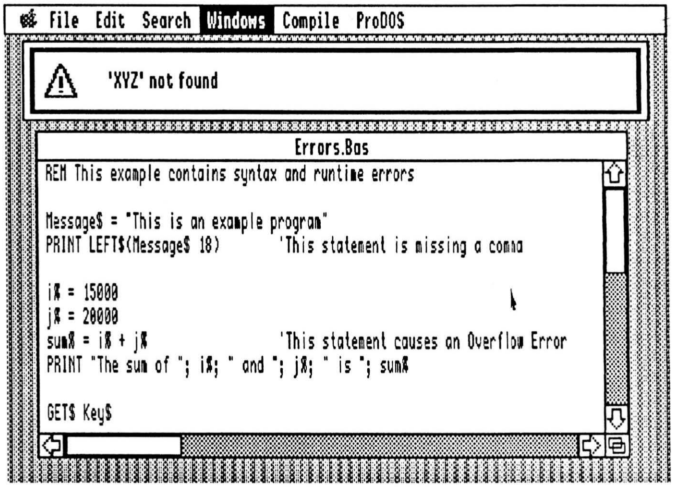
Editor Errors

To study how error messages are reported in the TML BASIC environment, first, close any open editing windows, and then open the file ERRORS.BAS from the PART1.EXAMPLES folder. We have intentionally placed several errors in the source code of this program so that you can see how the TML BASIC error reporting mechanism works.

The first type of error we will explore is an editor error. To cause an editor error select the **Find What** command from the **Search** menu (we will discuss this feature in greater detail in Chapter 4, but for now just follow along). The Find Dialog Box appears asking you to enter the text you wish to find. At this point, enter the string

"XYZ" without the double quotes, and then click the mouse on the Find button.

The string "XYZ" does not appear in the file ERRORS.BAS, so the editor reports this in the Error Dialog Box as seen below in Figure 3-3.



'XYZ' not found

```
REM This example contains syntax and runtime errors

Message$ = "This is an example program"
PRINT LEFT$(Message$ 18)        'This statement is missing a comma

i% = 15000
j% = 20000
sum% = i% + j%                  'This statement causes an Overflow Error
PRINT "The sum of "; i%; " and "; j%; " is "; sum%

GET$ Key$
```

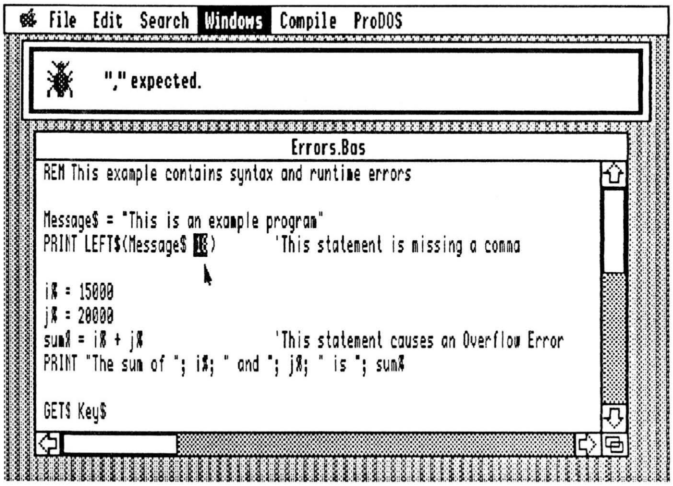
Figure 3-3
Error Dialog Box - Editor Error

Note the icon displayed in the left side of the Error Dialog Box. This icon indicates at the error was detected by the editor. To make the Error Dialog Box go away, simply click the mouse button or press any key on the keyboard. The editor detects and reports several different types of errors. For a complete list of the errors reported by the editor see Appendix A.

Compiler Errors

The ERRORS.BAS program also contains a syntax error in its source code - an illegal statement. To find this error simply select the **Check Syntax** command from the **Compile** menu. Figure 3-4 shows how the compiler reports syntax errors.

In this example, the message " ',' expected" is reported in the Error Dialog Box. This time a different icon appears – a small green bug. This icon indicates the error was detected by the compiler. In addition, the editor highlights the exact source code location of the encountered error in black, thus enabling you to fix the problem.



"," expected.

```
REM This example contains syntax and runtime errors

Message$ = "This is an example program"
PRINT LEFT$(Message$ 8)        'This statement is missing a comma

i% = 15000
j% = 20000
sum% = i% + j%                 'This statement causes an Overflow Error
PRINT "The sum of "; i%; " and "; j%; " is "; sum%

GET$ Key$
```

Figure 3-4
Error Dialog Box - Compiler Error

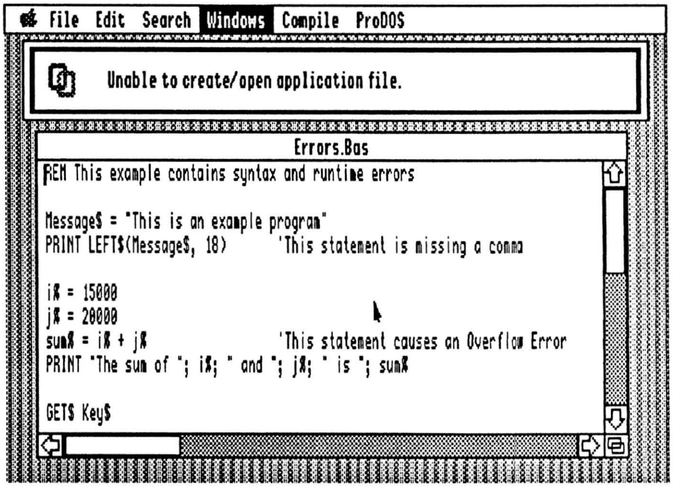
Again, click the mouse to make the Error Dialog Box go away and then enter the comma symbol where the compiler expected it. After correcting the syntax error, again select the **Check Syntax** command from the **Compile** menu. This time the compiler does not report any errors.

Do not compile this program using the **To Memory & Run** command yet, since there is another type of error we will discuss below.

Linker Errors

The final component of TML BASIC is the Linker. There exist only a few errors which the Linker can detect. One of these errors arises when the Linker attempts to write a program which has been compiled to disk, and an error occurs when writing to the disk. This error can occur when the disk is full or it has been write protected.

If you would like to see how the linker reports an error message, remove the floppy disk which contains the ERRORS.BAS source code file and write protect it. Then place the disk back in the disk drive and select the **To Disk** command from the **Compile** menu. After the Compiler successfully compiles the program it invokes the Linker to the program and attempts to write the stand-alone application file to disk. However, you have write protected the disk. Thus, the linker displays the error message seen in Figure 3-5. The icon used by Linker errors is two small chain links.



Unable to create/open application file.

```
REM This example contains syntax and runtime errors

Message$ = "This is an example program"
PRINT LEFT$(Message$, 18)      'This statement is missing a comma

i% = 15000
j% = 20000
sum% = i% + j%                 'This statement causes an Overflow Error
PRINT "The sum of "; i%; " and "; j%; " is "; sum%

GET$ Key$
```

Figure 3-5
Error Dialog Box - Linker Error

Runtime Errors

Actually, you might think of TML BASIC as having a fourth component – your program. When your program runs, it too can generate errors. For example, the program might attempt to add two numbers which cause an Overflow Error, or provide a value out of range thus causing an Illegal Quantity Error. These errors are called *runtime errors*. The built-in TML BASIC debugger is capable of detecting runtime errors and reporting them back to the TML BASIC environment so that you can modify your program accordingly.

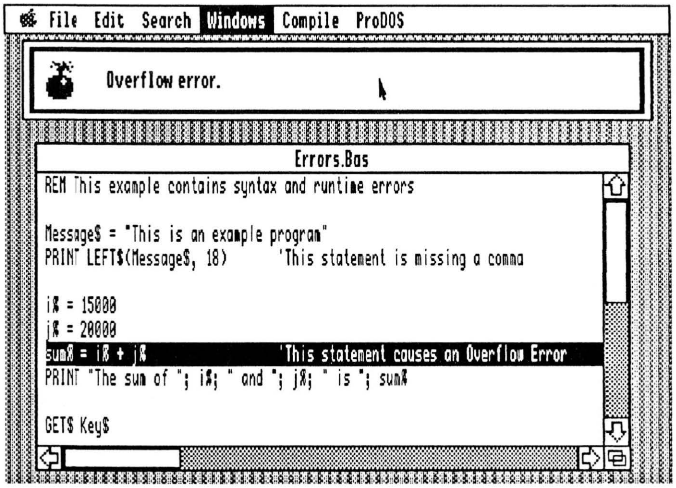
The TML BASIC debugger is available for debugging your programs only if you choose the **To Memory & Run** command from the **Compile** menu. In addition, you must instruct TML BASIC to turn on the debugger and generate the special debugging code needed to detect runtime errors by selecting the *Debug* option from the Preferences Dialog. The Preferences Dialog is displayed by choosing the **Preferences...** command from the **Compile** menu. The Preferences Dialog is discussed in detail in Chapter 6.

When you turn on the debugger by choosing the *Debug* option from the Preferences Dialog, the Compiler generates special code everywhere an error might potentially occur. Please note while this feature provides a powerful mechanism for developing programs, it does generate a significant amount making your programs larger and slower. After you have a program working correctly, it is generally a good idea to turn this option off.

Finally, the program ERRORS.BAS also contains a runtime error. To see how this feature works, make sure the the Debug option is turned on (it is on by default) in the Preferences Dialog, then select the **To Memory & Run** command from the **Compile** menu. If you successfully removed the syntax error in this program as described in the "Compiler Errors" section of this chapter, the compiler and linker should complete successfully. The TML BASIC environment then temporarily shuts down by hiding its menus and windows and runs the compiled program.

Unfortunately, this program has a runtime error in the seventh line. In this line, the value of *sum%* is set equal to the sum of the variables *i%* and *j%* (15,000 + 20,000). Because the variable *sum%* in an *integer variable,* the largest value it can store is 32,767. Thus, the value 35,000 *overflows* the capacity of *sum%*. The TML BASIC debugger detects this and aborts the execution of the running program. Upon returning to the TML BASIC environment, the line in which the error was detected is highlighted and the runtime error message "Overflow Error" is displayed in the Error Dialog Box as seen in Figure 3-6. In this example, the bomb icon is displayed to indicate the error is a runtime error.

The range and precision of numbers in TML BASIC is discussed in Chapter 7. In addition, each TML BASIC statement and function described in Chapter 10 lists the runtime errors that may possibly occur for each statement and function respectively.



Overflow error.

```
REM This example contains syntax and runtime errors

Message$ = "This is an example program"
PRINT LEFT$(Message$, 18)        'This statement is missing a comma

i% = 15000
j% = 20000
sum% = i% + j%                   'This statement causes an Overflow Error
PRINT "The sum of "; i%; " and "; j%; " is "; sum%

GET$ Key$
```

Figure 3-6
Error Dialog Box - Runtime Error

There are numerous errors that can occur when your program is running. Appendix A outlines each of these, along with a brief description of how each error might occur.

Just a Reminder...

In the course of editing, compiling, running programs and fixing errors, you may forget the last error that was detected and reported in the Error Dialog Box. If this happens, don't despair, it is possible to recall the Error Dialog Box to display the last error encountered. To do this, simply select the **Last Error** command in the **Windows** menu.

Finally, recall that all editor, compiler, linker and runtime errors are listed in Appendix A of this manual. Along with each error is a description of the error and usually some suggestions of how the error might have occurred and how to fix it.

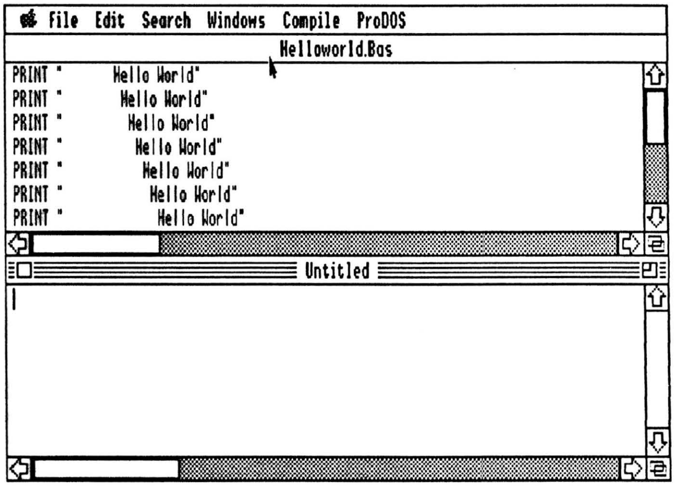
As always, before leaving this chapter, be certain to close all open windows without saving any changes to the files you might have made. To leave TML BASIC select the **Quit** command from the **File** menu.

Chapter 4

Advanced Program Editing

In Chapters 2 and 3, the principles of opening, closing, compiling and running programs were discussed. In this chapter, several additional TML BASIC features are introduced demonstrating TML BASIC's powerful editing commands. Additionally, the techniques for printing a file is discussed, and finally, you will learn about three ProDOS commands which may be issued from within TML BASIC.

Creating a New Editing Window



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello World"
PRINT "            Hello World"
```

Figure 4-1
Tiled Windows

Once again, boot your Apple IIGS using the *working copy* of TML BASIC and launch TML BASIC. Set-up your editing environment by first opening the HELLOWORLD.BAS example program and then selecting the **New** command from the **File** menu. Complete the set-up process by selecting the **Tile Windows** command from the **Windows** menu. Figure 4-1 illustrates the resulting screen's appearance after setting up your working environment.

Rather than using an example program alone to describe each of the edit features available to you in TML BASIC, we will create a new program using the empty window you just created. The result will be a complete program written entirely by borrowing source code from the HELLOWORLD.BAS example program.

Using the Clipboard

TML BASIC implements an editing feature called the *clipboard*. The clipboard is a temporary storage area for text and is used to store words, lines or entire portions of a program's source code. The fundamental idea of the clipboard allows you to borrow text from an existing program's source code for use in a different location within the same program or to use in an entirely different program.

TML BASIC allows you to place portions of text into the clipboard by using any one of its editing commands - cut, copy, paste, clear and select all - found in the Edit menu and discussed in the sections ahead. Prior to issuing any one of these four commands, the range of text to be placed in the clipboard must first be *selected*. Three methods are available to you when selecting text - *dragging*, *shift-clicking* and *double-clicking*. One method may be more appropriate than the other two depending upon the range of text to be selected.

Before discussing the three different means of selecting text, we will define the *current insertion point*. The current insertion point is located where the cursor appears after clicking the mouse once in the active window. Using the location of the current insertion point you can begin entering or deleting characters, or you can use the location for marking the beginning of text to be *selected*.

Dragging is the easiest way to select text, and can be accomplished by moving the cursor on the screen to the beginning of the text you wish to select, then click and hold the mouse button down while dragging the cursor to the end of the text to be selected. You will notice the range of selected text appears in black (inverted) type. Dragging allows you to be extremely particular about the range of text you select.

Secondly, you can select a range of text by *shift-clicking*. To shift-click, move the cursor to the beginning of the text you wish to select and click the mouse once. Next, move the cursor to the end of the text you wish to select, then hold down the *Shift Key* and click the mouse button once to select all of the text between the first and second mouse clicks. This method is best used to select a large portion of a program's source code for placement in the clipboard.

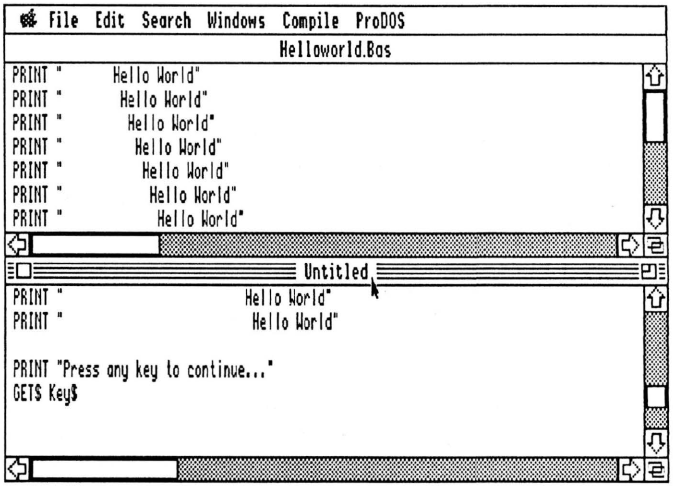
Thirdly, to select text by *double-clicking*, position the mouse over the text you would like to select, and quickly press the mouse button twice. Double-clicking will select the entire word which appears under the cursor. Double-clicking is most useful when only a single word is to be placed in the clipboard.

Editing Commands

At the heart of TML BASIC is its ability to assist you in writing programs. Unlike conventional editors, TML BASIC provides a full-screen editing environment enabling you to both write programs more quickly and make necessary source code changes with little difficulty.

With HELLOWORLD.BAS selected as the active window, issue the **Select All** command from the **Edit** menu. Notice that the entire contents of the HELLOWORLD.BAS window is inverted and placed in black type. Remember, inverted text represents the range of selected text. Now, issue the **Copy** command from the **Edit** menu. You will not see any changes on the screen, but rest assured the selected range of text now exists in TML BASIC's clipboard and may be used to assist in creating a new program.

Position the cursor anywhere in the "Untitled" window and click the mouse once, resulting in the "Untitled" window becoming the active window. Now, move the cursor to the first line in the "Untitled" window and click the mouse once to create the current insertion point. Issuing the **Paste** command from the **Edit** menu at this time results in the clipboard's present contents being *pasted* into the "Untitled" editing window beginning at the current insertion point. Figure 4-2 illustrates the result of performing this **Select All/Copy/Paste** command.



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello World"
PRINT "            Hello World"
```

```
PRINT "                        Hello World"
PRINT "                         Hello World"

PRINT "Press any key to continue..."
GET$ Key$
```

Figure 4-2
Pasting Source Code

Unlike the **Copy** command, the **Cut** command results in the selected range of text being completely removed from its original location in a program's source code and placed into the clipboard for use elsewhere by issuing the **Paste** command. If you had issued the **Cut** command in place of the **Copy** command in the last example, the HELLOWORLD.BAS editing window would appear empty with its entire contents placed in TML BASIC's clipboard.

That brings to mind one last editing command - the **Clear** command. Issuing the **Clear** command after selecting a program's entire range of text results in the active window's contents being *erased*. The program's source code is not placed in the clipboard and you would have to issue the File menu's **Revert** command in order to restore the program's source code. The **Clear** command is useful when you do wish to erase a selected range of text or an entire window's text and without disturbing the clipboard's contents.

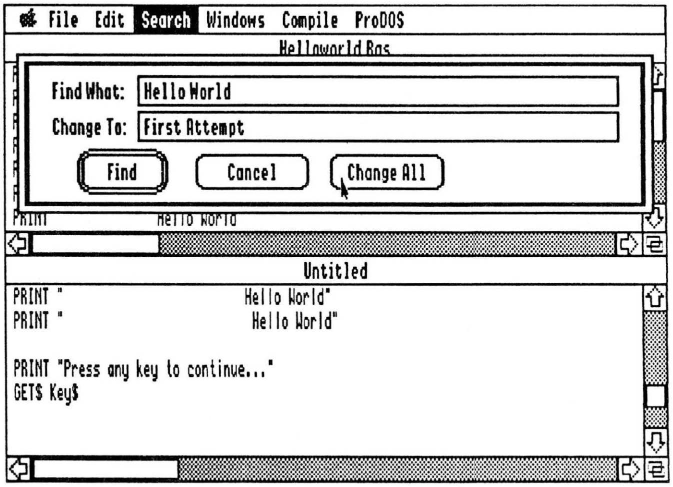
Obviously, TML BASIC's editing commands are not limited to selecting an entire program's source code. Using any one of the three techniques discussed in the previous section for selecting text (dragging, shift-clicking and double clicking) enables you to cut, copy, paste and/or clear any range of text within a program.

Note that the clipboard can only hold one piece of information at a time, so everytime you cut or copy from a window, any information previously in the clipboard is replaced. Each time you paste from the clipboard, however, a duplicate copy of the information remains in the clipboard.

Find and Change

After pasting the HELLOWORLD.BAS program's entire source code into the "Untitled" window, the next step in our exercise is to change each occurrence of the character string "Hello World" to "First Attempt" in the "Untitled" window. One way to do this, of course, would be to move the cursor to the end of the each occurrence of the string "Hello World", and then backspace over each "Hello World" string and re-type "First Attempt". This could be an extremely cumbersome and time consuming task, so let's investigate TML BASIC's ability of finding a string of text and replacing it with another string of text.

The TML BASIC **Search** menu contains three commands for locating and changing text in an active window. The first command is the **What to Find...** menu item. Choosing this command displays the TML BASIC Find Dialog Box shown in Figure 4-3. The Find Dialog Box request two entries be made – *Find What* and *Change To* , and three buttons – *Find, Cancel* and *Change All.*



```
PRINT "                    Hello World"
PRINT "                     Hello World"

PRINT "Press any key to continue..."
GET$ Key$
```

Figure 4-3
Find Dialog Box

The *Find What* item is where the string to be searched for is specified. For our example, you should enter the string "Hello World" here. If all you wanted to do was find the next occurrence of this string then you would press the *Find* button in the Find Dialog Box. The dialog would go away, and TML BASIC would search the program contained in the active window, beginning at the *current insertion point* for the next occurrence of "Hello World". However, in our example, you also need to change the string "Hello World" to "First Attempt" once it is found. Thus, you should enter the string "First Attempt" in the *Change To* text edit item, and then press the *Find* button.

After pressing the *Find* button, the string "Hello World" is found, and is automatically selected by inverting the string in black. To change the string to "First Attempt", as you specified in the Find Dialog Box's *Change To* text edit item, simply issue the **Change then Find** command from the **Search** menu. After changing the string, TML BASIC proceeds to find the next occurrence of "Hello World". Once every occurrence of the "Hello World" string has been changed using this method, TML BASIC reports that the string cannot be found in an Error Dialog Box.

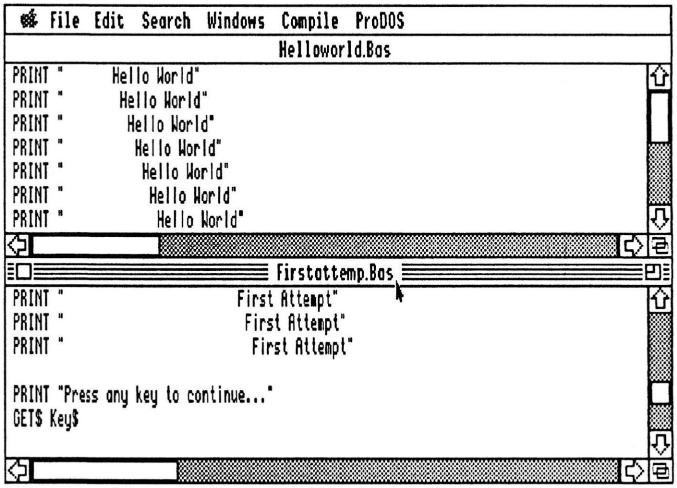
Also, you could save yourself the time of locating and changing each occurrence of the "Hello World" string by pressing the *Change All* button in the Find Dialog Box.

One final change you should make is to save the program to disk under the new name of FIRSTATTEMP.BAS. Selecting the **Save As** command from the **File** menu results in the Save File Dialog Box displayed on your screen. The Save File Dialog

Box contains buttons allowing you to determine both the disk and folder you wish to save the new program in, as well as the name it will be given for future use.

The disk and folder will already by identified as the TML BASIC disk and the PART1.EXAMPLES folder respectively, so all you need to do is enter the name FIRSTATTEMP.BAS at the *Save document as:* insertion point in the dialog.

The result of these changes, as well as the copy and paste commands performed, is a new program, which for the most part does exactly what the HELLOWORLD.BAS program does. Figure 4-4 illustrates the screen's appearance as a result of creating the new FIRSTATTEMP.BAS program.



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello World"
PRINT "            Hello World"
```

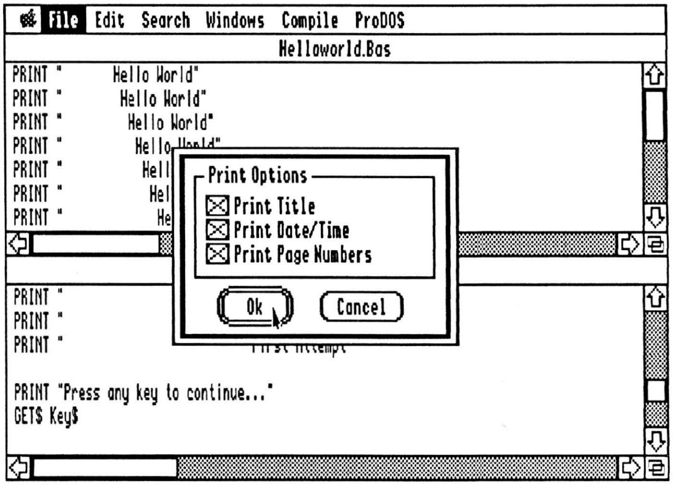
```
PRINT "                       First Attempt"
PRINT "                        First Attempt"
PRINT "                         First Attempt"

PRINT "Press any key to continue..."
GET$ Key$
```

Figure 4-4
New File - FIRSTATTEMP.BAS

Printing

TML BASIC can print the contents of the active editing window using the **Print** command from the **File** menu. When selecting this command, the text in the topmost window is printed to the printer using the current printing options. TML BASIC prints to either of the serial ports (slots 1 and 2) directly. Thus, TML BASIC can print to any serial printer, such as the ImageWriter, or to any parallel printer connected to an interface card in either slot 1 or 2. TML BASIC provides three commands in the **File** menu for controlling the way a file is printed: **Print Options, Page Setup** and **Chooser**.

The **Print Options** command is used to define the information printed on a page. When selecting this command the Print Options Dialog Box is displayed as shown in Figure 4-5.



```
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World"
PRINT "           Hello
PRINT "            Hell
PRINT "             Hel
PRINT "              He
```

Print Options
- ☒ Print Title
- ☒ Print Date/Time
- ☒ Print Page Numbers

```
PRINT "
PRINT "
PRINT "

PRINT "Press any key to continue..."
GET$ Key$
```

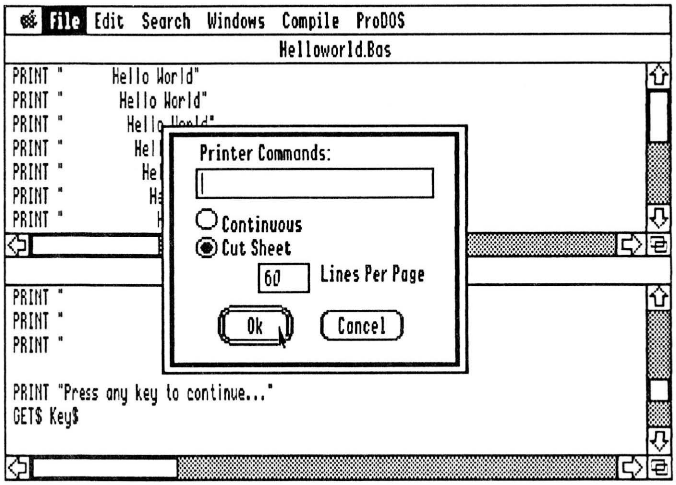
Figure 4-5
Print Options Dialog Box

When TML BASIC prints a file to the printer, it optionally prints a header across the top of every page. The header can include the name of the file (Print Title), the current date and time (Print Date/Time), and page numbers (Print Page Numbers). If an option is checked, TML BASIC prints the corresponding information in the header. If none of the options are selected, a header is not printed.

The **Page Setup** menu command displays the Page Setup Dialog Box (Figure 4-6) when selected. This dialog is used to configure the way TML BASIC prints a page. There are two options: Continuous and Cut Sheet. If Continuous is selected, a header is only printed on the first page, and no blank lines are printed at the end or beginning of a piece of paper. This option maximizes the number of lines that can be printed on a page. However, if the paper is misaligned, a line of text may print on the perforation in the paper.

If the Cut Sheet option is used, a header is printed at the top of every page, and blank lines are printed at the end and beginning of every page. When this option is selected, the number of lines per page must be set. The default setting is for standard 8 1/2 by 11 inch paper.

Finally, the Page Setup Dialog Box allows you to enter a special character sequence representing a Printer Command. The character sequence is sent to the printer before printing every file. The Printer Command can be used to instruct a printer to use a special built in font or font size, page size, etc. In order to send a control character to the printer use the caret character (^) followed by the appropriate letter that defines the control character. For example, ^? sends an ASCII ?? (an escape character). See Appendix E for these codes.



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        Hell
PRINT "         Hel
PRINT "          He
PRINT "           H
PRINT "           H

PRINT "
PRINT "
PRINT "

PRINT "Press any key to continue..."
GET$ Key$
```

Printer Commands:

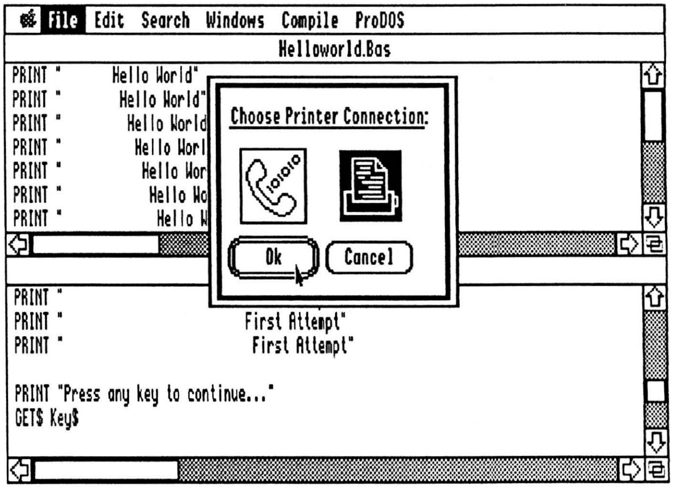
○ Continuous
[✓ ● Cut Sheet]
60 Lines Per Page

Ok   Cancel

Figure 4-6
Page Setup Dialog Box

The **Chooser** menu item allows you to specify to TML BASIC which of the two serial ports (printer port or modem port) your printer is connected. If you have changed the Apple IIGS Control Panel so that either slot 1 or slot 2 does not use the built-in port, but rather a card in that slot, TML BASIC will obey this change so that you can use a parallel printer with an appropriate card.

This menu command displays the Choose Printer Connection Dialog Box as shown in Figure 4-7. The Printer icon indicates the built-in printer port or slot 1, and the phone icon indicates the built-in modem port or slot 2. Click the mouse on the icon which has the printer connected.


File Edit Search Windows Compile ProDOS
Helloworld.Bas
Choose Printer Connection:
Ok
Cancel

```
PRINT "        Hello World"
PRINT "         Hello World"
PRINT "          Hello World
PRINT "           Hello Worl
PRINT "            Hello Wor
PRINT "             Hello Wo
PRINT "              Hello W
```

```
PRINT "
PRINT "                              First Attempt"
PRINT "                               First Attempt"

PRINT "Press any key to continue..."
GET$ Key$
```

Figure 4-7
Choose Printer Connection Dialog Box

In addition to selecting the printer port, you should also make sure the selected port is properly configured for the type of printer you have. You can change the configuration of the serial ports using the Apple IIGS Control Panel. If you are not familiar with this operation see Appendix A of your *Apple IIGS Owner's Guide.*

All of your selections for printing configurations are saved to the TMLBASIC.OPTS file so that you do not need to specify your selections every time you use TML BASIC.

After selecting the correct printer configuration, click the mouse once on the new FIRSTATTEMP.BAS program, making it the active window if it is not already, and then select the **Print** command from the **File** menu.

While TML BASIC is printing the contents of the active window a small Print Dialog Box is displayed in the middle of the editing screen. Before issuing the **Print** command, be certain your printer is *on* and *selected*. To cancel printing at any time, press the mouse button or any key on the keyboard.

The Preferences Dialog

The **Preferences** command found in the **Compile** menu allows you to customize the way you use TML BASIC.

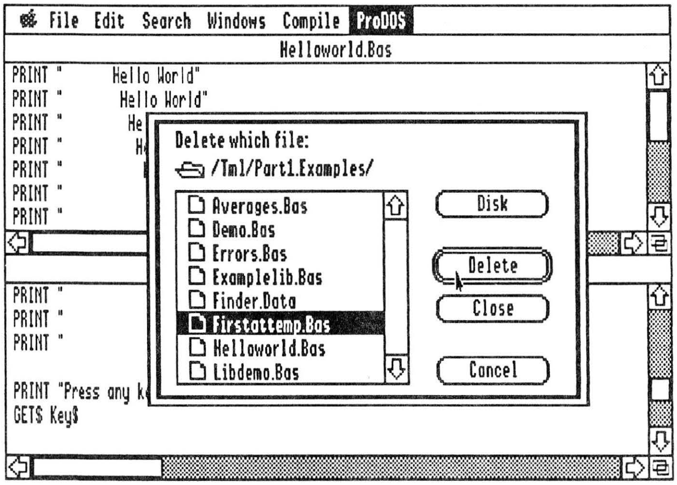
The Preferences Dialog contains valuable information allowing you to customize TML BASIC's functionality to meet your particular programming needs. The information contained in the dialog is extremely important and ranges from allowing you to adjust TML BASIC's tab settings to freeing up memory space. Chapter 6 includes a detailed description of each item found in the Preferences Dialog Box and should be studied carefully before you begin programming.

ProDOS Commands in TML BASIC

TML BASIC provides access to three ProDOS commands without requiring you leave the TML BASIC environment. To see these commands, pull-down the **ProDOS** menu. You will see the **Rename...**, **Delete...** and **Transfer...** menu items. Use of these three ProDOS commands while working within TML BASIC will save you valuable programming time otherwise lost if you were required to leave TML BASIC everytime you wished to issue one of the these commands.

Selecting any one of the three ProDOS commands results in a modified version of the Get File Dialog Box displayed. The modified Get File Dialog Box's *Open* button will be changed to an appropriate title matching the command being performed.

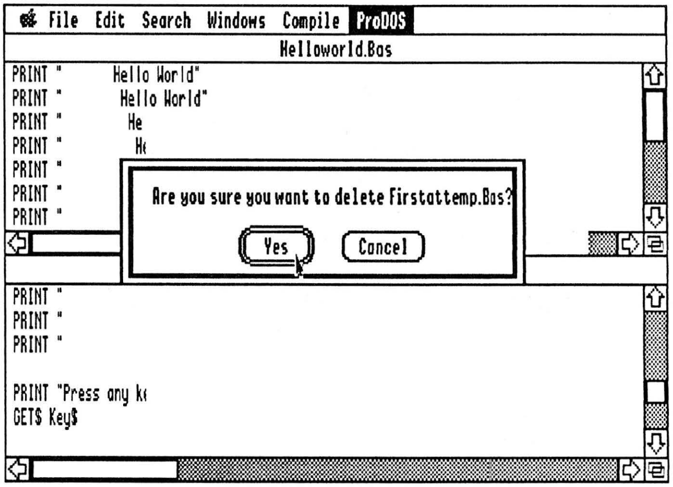
Select the **Delete...** command from the **ProDOS** menu. The Get File Dialog Box is displayed as shown in Figure 4-8 allowing you to select a file to be deleted. Locate the file FIRSTATTEMP.BAS you created earlier in this chapter and delete it.



```
PRINT "      Hello World"
PRINT "       Hello World"
PRINT "        He
PRINT "         H
PRINT "
PRINT "
PRINT "

PRINT "
PRINT "
PRINT "

PRINT "Press any k
GET$ Key$
```

Figure 4-8
Get File Dialog Box - Delete

Before TML BASIC erases the file from the disk it displays the Delete Confirmation Dialog Box (Figure 4-9). This dialog gives you one last chance to prevent deleting the wrong file. In this particular case we are certain the FIRSTATTEMP.BAS file should be deleted, so you should click the mouse on the *Yes* button to proceed with deleting the file from your TML BASIC working disk.



```
PRINT "       Hello World"
PRINT "        Hello World"
PRINT "         He
PRINT "          H(
PRINT "
PRINT "
PRINT "
```

Are you sure you want to delete Firstattemp.Bas?

```
PRINT "
PRINT "
PRINT "

PRINT "Press any k(
GET$ Key$
```

Figure 4-9
Delete Confirmation Dialog Box

The second ProDOS command available to you in TML BASIC allows you to rename a file from within TML BASIC. After selecting a file to be renamed, TML BASIC displays a Rename Dialog Box allowing you to change the current name of the selected file. In this respect, each of the three ProDOS commands are alike - you are always provided a second opportunity to prevent the inadvertent consequences of deleting, renaming or transferring control to the wrong file.

The third ProDOS command available in TML BASIC is the **Transfer** command. The **Transfer** command is used to leave TML BASIC permanently and transfer control to another ProDOS application (even one you might have created with TML BASIC). After transferring control to the chosen application, you can only return to TML BASIC by first quitting the newly selected application and then re-launching TML BASIC from the IIGS Finder.

Chapter 6 includes a complete reference to all of TML BASIC's commands and features and should be referred to for further information about each of the three

ProDOS commands available from within TML BASIC.

In this chapter we have explained some of the advanced editing commands available to you while creating a new program similar to the HELLOWORLD.BAS program used throughout Chapter's 2 through 4. You also learned about printing program listings, and how to issue three powerful ProDOS commands without leaving TML BASIC. Customizing the TML BASIC environment via the Preferences Dialog Box was briefly discussed with a forward provided to Chapter 6.

As always, be certain to close all open editing windows used in this chapter and leave TML BASIC by issuing the **Quit** command.

The next chapter introduces a few of the fundamental statements which are part of the TML BASIC language. A thorough understanding of TML BASIC's working environment, as discussed up to this point, will increase your performance at the keyboard when following the discussion in Chapter 5.

Chapter 6 includes a brief description of every TML BASIC feature and should be used as a reference in your programming efforts.

Chapter 5

Your First Program

This chapter assumes you have familiarized yourself with TML BASIC's programming environment. If you are not already familiar with how TML BASIC works - in particular the techniques for editing, compiling and running a program - you should take the time to read Chapter's 1 through 4 before proceeding.

TML BASIC can be used to create two different types of programs: *Textbook* programs and *Toolbox* programs. Textbook programs represent "traditional" style programs created using the Apple IIGS text screen. We use the term *textbook* because these are the types of programs you are likely to find in most general BASIC programming textbooks. Toolbox programs, on the other hand, are those which make use of the Apple IIGS Toolbox. These programs operate within the Apple IIGS's Super Hi-Res Graphics screen and are usually event-driven applications which use the mouse. This chapter addresses only textbook programs. Part III of this manual, "Toolbox Programming", documents the Toolbox and how to write Toolbox programs.

Although TML BASIC provides a large number of predefined programming statements and functions, you will find only a small number of these statements and functions necessary to begin programming. For a complete reference to every statement and function included in TML BASIC reference Chapter 10.

Statements introduced in this chapter:

REM
LET
PRINT
GET$
INPUT

The First Program

To begin this discussion of textbook programming, launch TML BASIC from your working disk and open the file AVERAGES.BAS located in the PART1.EXAMPLES folder. If you have not already made a working copy of TML BASIC, you should do so now before proceeding in order to protect your distribution disk from possible damage. Additionally, if you are not familiar with how to launch TML BASIC and open, edit, compile and run programs, refer to Chapter's 1 through 4 of this manual.

After you have opened the AVERAGES.BAS program, the following source code should appear in the editing window:

```
REM A program to compute the average of three numbers
LET Avg = (43 + 27 + 23) / 3
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

Run the AVERAGES.BAS program by selecting the **To Memory & Run** command from the **Compile** menu. The program is immediately compiled and then run by TML BASIC. The output from this program should appear similar to the following line of text on the text screen:

```
The average of the three numbers is 31
```

Press any key on your keyboard and the program completes its execution. The TML BASIC environment reappears with the open window still containing the AVERAGES.BAS program's source code. Now let's examine how this program actually works.

The program consists of four lines of source code. Each line contains a *statement*. Statements are the fundamental component of a TML BASIC program used to instruct the computer what actions to perform. Each statement begins with a special TML BASIC word called a *reserved word*. The reserved word indicates what kind of statement is on the line. A list of all the TML BASIC reserved words can be found in Chapter 7, Table 7-2.

If you are familiar with other implementations of the BASIC language you will immediately notice there are no *line numbers* in a TML BASIC program. Line numbers have traditionally been part of the BASIC language because the editors used with these older implementations required them. In addition, line numbers were used in some BASIC statements. TML BASIC, on the other hand, uses *alphanumeric labels* instead of line numbers. Alphanumeric labels are discussed in detail in Chapter 7.

The REM Statement

The first statement used in the AVERAGES.BAS program is the REMark statement. The REM statement is used in a TML BASIC program to include notes about a program's purpose, what it does, how it works or any other information you find useful to describe the program. It is also good programming practice to include REM statements in your programs in order to "document" their actions.

The REM statement does not instruct the computer to perform any specific action. In fact, TML BASIC ignores the remainder of a line containing a REM statement.

The AVERAGES.BAS program includes one REM statement used to describe the program's purpose.

TML BASIC offers an alternative to the REM statement called the *comment*. A comment behaves exactly like the REM statement except that it consists only of the single quote (') character followed by appropriate documentation. For example, the AVERAGES.BAS program can be rewritten as:

```
'A program to compute the average of three numbers
LET Avg = (43 + 27 + 23) / 3
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

The important difference between the REM statement and a comment is that a comment is not a statement. If a program has a REM statement which appears on the same line after another statement, a colon must be used to separate the two statements (colons can be used to separate several statements on the same program line). However, a comment does not require a preceding colon since it is not a statement. Consider the following example:

```
LET Avg = (43 + 27 + 23) / 3     :REM Compute the average
LET Avg = (43 + 27 + 23) / 3     'Compute the average
```

The first programming line above illustrates two statements included on the same line and separated by a colon. The second line, however, contains only one program statement with a proceeding comment to document the line's purpose.

The LET Statement

The LET statement, also called the *assignment* statement, is used to assign a value to a variable. Following the reserved word LET is a variable name followed by an equal sign and then an expression. The variable on the left side of the equal sign is given the value of the expression on the right side of the equal sign.

A *variable* is a named entity which stores a numeric or string value. The variable's value can change during execution of a program by using the LET statement. The variable name is any sequence of alphanumeric characters that begins with a *letter* and does not spell any of the TML BASIC reserved words. A variable name may be of any length and all characters are significant. If the variable stores a string value, its name must end with the dollar sign character ($). For more information about variables see the section "Variables" in Chapter 7. In the AVERAGES.BAS program, *Avg* is a numeric variable and *Key$* is a string variable.

An *expression* represents a value. An expression is made up of *operands* combined with *operators* which produce a value when evaluated during execution. Operators are special symbols representing a particular operation to be performed. For example, the LET statement used in the AVERAGES.BAS program contains both the addition operator (+) and the division operator (/). Operands are constants, variables and function calls that operators work on. In the LET statement, the constants 43, 27, 23 and 3 are operands. Again, more information about expressions can be found in Chapter 7, in the section titled "Expressions".

The following example shows how the AVERAGES.BAS program can be rewritten to use several LET statements and variables within an expression.

```
REM A program to compute the average of three numbers
LET Count = 3
LET Number1 = 43
LET Number2 = 27
LET Number3 = 23
LET Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

As previously mentioned, every TML BASIC program statement must begin with a reserved word. The only exception to this rule is the LET statement. In the case of the LET statement, the reserved word LET is optional and need not appear. Thus, TML BASIC assumes any statement that begins with a *variable* is, in fact, a LET statement.

The following example shows how the AVERAGES.BAS program can be rewritten using the LET statement without the reserved word LET.

```
REM A program to compute the average of three numbers
Count = 3
Number1 = 43
Number2 = 27
Number3 = 23
Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

Enter these changes in the AVERAGES.BAS program and try running the program to see for yourself how the LET statement operates.

The PRINT Statement

The PRINT statement displays text on the Apple IIGS text screen. The PRINT statement is used to print the values of numeric and string expressions. The PRINT statement may contain any number of expressions separated by either a comma or semicolon. Each expression is called a *print item*. Actually, multiple expressions can be separated by spaces, but it is good programming practice to use either a comma or a semicolon to clearly show that more than one expression is included in the PRINT statement.

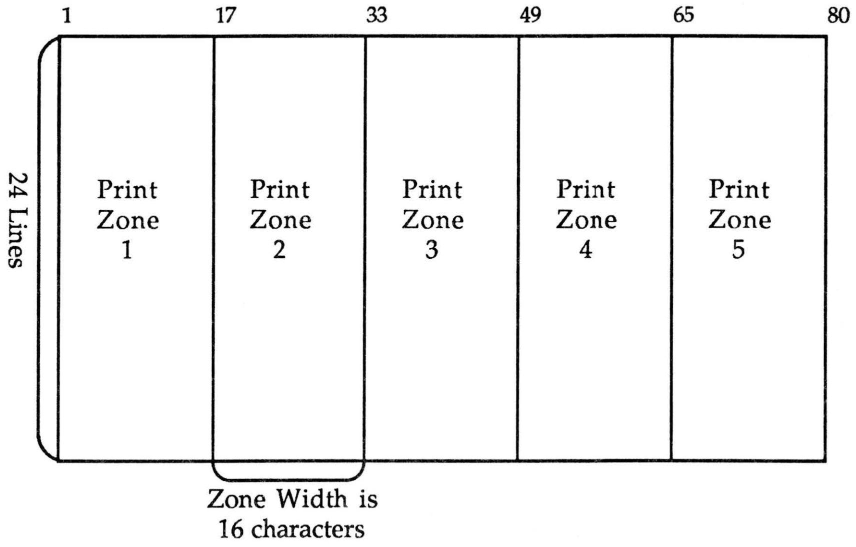
The PRINT statement used in the AVERAGES.BAS program contains two print items separated by a semicolon. The first item is the string constant

    "The average of the three numbers is "

and the second item is the numeric variable *Avg*.

When a string expression appears in a PRINT statement, the exact characters in the string are displayed to the text screen at the current text location (the location of the cursor). When a numeric expression is printed, the binary representation of the numeric value is first converted to a string and then displayed at the current text location.

When using the semicolon as a separator between multiple expressions in a PRINT statement, TML BASIC positions the cursor immediately following the last character displayed. Thus, the next expression is displayed adjacent to the previous print item. Using a comma as a separator causes TML BASIC to perform a tab operation before the next print item is displayed. The tab width of the PRINT statement is 16 characters. The spaces between each tab are called a *print zone*. The following diagram illustrates how the 80 columns of a text screen are divided into five print zones.

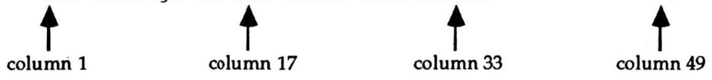


If the PRINT statement from the AVERAGES.BAS program is rewritten to use the comma separator as shown below,

```
PRINT "The average of the three numbers is ", Avg
```

the output will be displayed as follows:

```
The average of the three numbers is            31
```



Because the position of the text cursor is at column 36 after printing the string constant, the comma causes the cursor to tab to the next print zone beginning in column 49.

After all print items within a PRINT statement have been displayed as output, the text cursor is moved to the first column of the next line. If the cursor is on the last line of the screen, the entire contents of the screen is scrolled up one line. Thus, a PRINT statement containing no print items will display one blank line.

In some cases, a program may not want the PRINT statement to advance the text location to the next line after it has displayed all of its print items. Whenever a PRINT statement ends with a comma or a semicolon, the PRINT statement will not advance to the next line. For example, the PRINT statement in the AVERAGES.BAS program can be rewritten using two PRINT statements, and have the output displayed on one line as before.

```
REM A program to compute the average of three numbers
LET Avg = (43 + 27 + 23) / 3
PRINT "The average of the three numbers is ";
PRINT Avg
GET$ Key$
```

The TML BASIC language has several other variations and functions which work in conjunction with the PRINT statement. These include the PRINT USING statement as well as SHOWDIGITS, SPC and TAB. These are advanced functions described in detail in Chapter 10.

The GET$ Statement

The GET$ statement is used to assign a single character from the keyboard to a string variable, without displaying it on the screen and without requiring the Return Key be pressed.

Examining the AVERAGES.BAS program you will notice the GET$ statement is used without the character read from the keyboard used anywhere else in the program. The reason for the GET$ statement appearing in this program (as it does in the other example programs found on disk) is to temporarily halt execution of the running program before control is returned to the TML BASIC environment. If this statement did not appear here, the output of the program would be displayed on the screen and control would return to TML BASIC so fast you would not be able to see the program's output.

Delete the GET$ statement from the AVERAGES.BAS program and then compile the program to see how quickly the program returns control to TML BASIC after completing execution.

Of course, you can use the GET$ statement throughout your program for the specific purpose of receiving input from the user and then acting upon it. However, for the purpose of this chapter's discussion, you need only realize the statement is used to temporarily keep a program from returning control to TML BASIC so that you can see the program's output.

The INPUT Statement

The AVERAGES.BAS is a fine first program, but it does have one rather serious flaw: it only averages the three numbers 43, 27 and 23. After running this program a few times it becomes obvious the average of these three numbers is 31. In this respect, the program would be much more useful if it could average any three numbers as input by the user.

The INPUT statement is TML BASIC's means of obtaining one or more numeric or text values entered at the keyboard. When the INPUT statement is executed, TML BASIC accepts a value entered from the keyboard and assigns it to the first variable in the INPUT statement. Consider the following variation of the AVERAGES.BAS program:

```
REM A program to compute the average of three numbers
Count = 3
INPUT Number1
INPUT Number2
INPUT Number3
Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

When the INPUT statement is executed, a question mark (?) is displayed on the screen indicating the program is waiting for input. Try entering this program and compiling it with TML BASIC to see how the INPUT statement works.

The INPUT statement can now accept several values at a time by listing several variables in the statement separated by commas. For example, the above program could be rewritten with only one INPUT statement as follows:

```
REM A program to compute the average of three numbers
Count = 3
INPUT Number1, Number2, Number3
Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

When more than one variable is listed in an INPUT statement, each of the values entered at the keyboard must be separated by a comma or a Return key. If a Return key is entered variables still exist which have not been given values, TML BASIC displays two question marks (??) indicating more data is required by the INPUT statement.

The INPUT statement may also contain a string which is displayed as the input prompt instead of the normal question mark. The string must appear immediately after the reserved word INPUT and must be a string constant and *not* a string variable or expression. The following example shows how this variation of the INPUT statement can be used in the AVERAGES.BAS program.

```
REM A program to compute the average of three numbers
Count = 3
INPUT "Enter three numbers: "; Number1, Number2, Number3
Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

Of course, the INPUT statement can be used to input string variables as well as numeric variables by simply listing a string variable as an argument to the INPUT statement as follows:

```
INPUT "Enter three names: "; Name1$, Name2$, Name3$
```

Multiple Statements

TML BASIC programs consist of program lines where each line contains a program statement. If you desire, it is actually possible to include several statements on a source code line by separating the statements with a colon (:) character. The only limit on the number of statements that may appear on a line is the restriction that TML BASIC source code lines may not exceed 255 characters.

The following example illustrates how our modified version of the AVERAGES.BAS program can be rewritten to include four LET statements on a single line. Note that between each statement is a colon to separate the two adjacent statements.

```
REM A program to compute the average of three numbers
Count = 3: Number1 = 43: Number2 = 27: Number3 = 23
Avg = (Number1 + Number2 + Number3) / COUNT
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

While this technique reduces the number of lines in a program, it makes the program harder to read. Unlike interpreted BASIC implementations, TML BASIC imposes no overhead for using extra program lines. In fact, you can even include blank lines in TML BASIC without creating errors.

Summary

This chapter introduced some of the most fundamental statements used by any TML BASIC program. They are the REM, LET, PRINT, GET$ and INPUT statements.

Although the example program used in this chapter's discussion seems relatively simple, you can probably imagine how complicated programs are capable of being written with only these statements. Even so, your programming skills are certain to increase rapidly, thus demanding more powerful statements to include in your programs.

As mentioned in the onset of this chapter's discussion, Chapter 10 is a complete guide to all of the statements and functions available to you in TML BASIC. Reference Chapter 10 at any time while programming for guidance in using each statement and function.

Chapter 6
TML BASIC Menu Reference

This chapter provides a complete reference to the commands available and contained in each of TML BASIC's seven menus. TML BASIC's seven menus are the **Apple**, **File**, **Edit**, **Search**, **Windows**, **Compile**, and **ProDOS** menus. Recall that most menu commands can be issued by entering *command-key equivalents* rather than clicking the mouse on the menu and releasing it over the menu command. If a menu command has a command-key equivalent, it is shown in the menu command's heading below. A discussion of command-key equivalents is provided at the end of this chapter.

The Apple Menu

The **Apple** menu is a standard menu for Apple IIGS desktop applications such as TML BASIC, and is always the first menu in the menu bar. In TML BASIC, the **Apple** menu has two parts: the **About TML BASIC...** command and the list of installed new desk accessories (NDAs) available in TML BASIC. Because, the list of desk accessories depends upon which desk accessories are installed on your particular system disk, Figure 6-1 may not match your menu exactly.

About TML BASIC... ?

The **About TML BASIC...** menu item displays the About BASIC Dialog Box. The dialog contains the TML BASIC logo, TML Systems' address and phone number. More importantly, the version of TML BASIC you are using and TML Systems' copyright notice also appears in the About BASIC Dialog Box.

Desk Accessories

The desk accessory menu items represent each of the NDAs installed on your system disk. Recall that desk accessories must be properly installed on your bootable system disk to be available. For a desk accessory to be properly installed, it must be in the DESK.ACCS folder which is in the SYSTEM folder. Selecting a desk accessory name from the Apple menu will cause that desk accessory's window to be opened on the TML BASIC desktop.

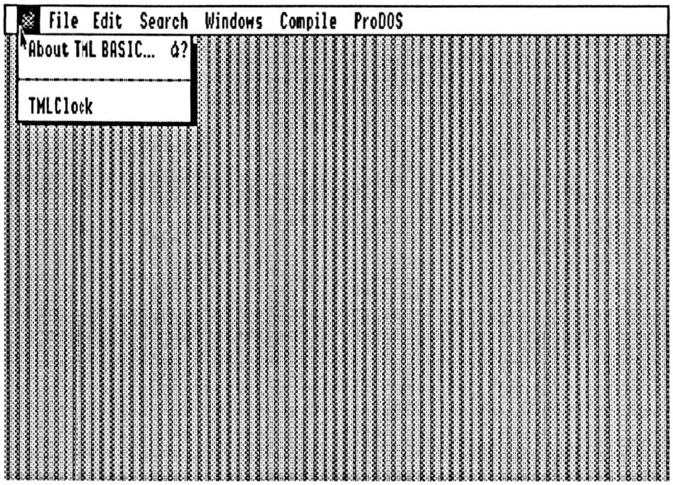


Figure 6-1
Apple Menu

The File Menu

The **File** menu contains the various file related commands (Figure 6-2) in TML BASIC. The menu items are grouped into three basic categories: accessing disk files, printing, and exiting TML BASIC. Following is a description of each menu item contained in the **File** menu.

New ⌘N

This item opens a new ("Untitled") window. The new window becomes the active window ready for editing. If four windows have already been opened, the maximum allowed by TML BASIC, then this item is disabled.

Open ⌘O

The **Open** menu item displays the Open File Dialog Box (Figure 6-3) allowing you to select a file for editing or compiling. This menu item is disabled if four windows are already open.

Close ⌘K

This menu item closes the active (topmost) editing window. If the source code contained in the active window has had changes made to it since last opened, you are prompted to save the changes you have made, and if the active window is untitled, you are asked to name the file.

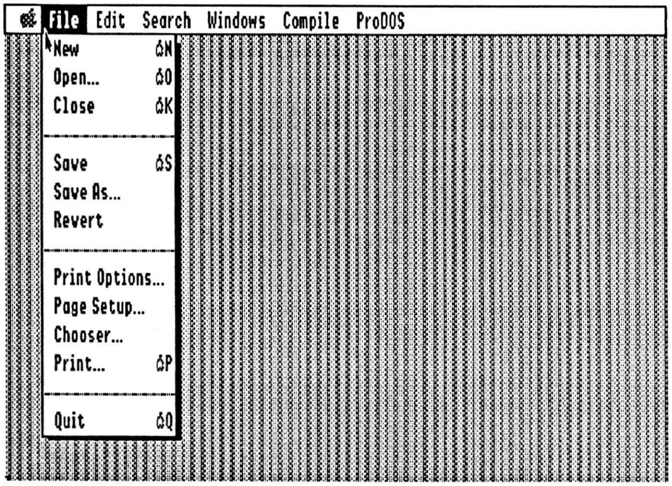


Figure 6-2
File Menu

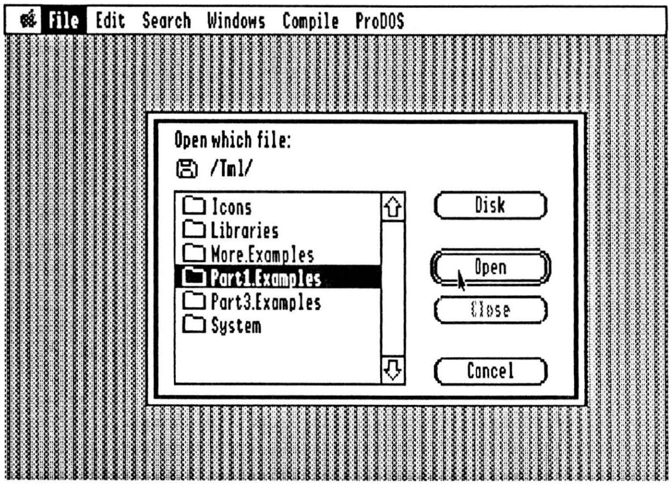


Figure 6-3
Open File Dialog Box

Save ⌘S

The **Save** menu item saves the contents of the active window. If the window is already associated with a disk file, the original file on disk is overwritten by the contents of the current window. If the window is untitled, you are prompted with the Put File Dialog Box to name the window.

Save As...

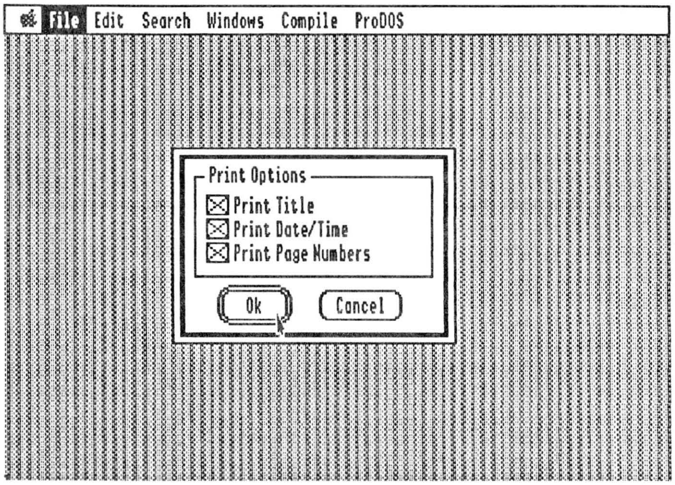
Selecting this menu item allows you to save the contents of the active editing window to a new disk file. To do this, you are again prompted with the Put File Dialog Box to name a file for this window. If the filename you choose already exists in the specified subdirectory, you will be warned of this and asked if you wish to replace the existing file.

Revert

This menu item will cause all of the editing changes you have made to be replaced with the most recently saved version of the file. You will be asked to confirm this choice before the operation is performed.

Print Options...

This menu item displays the Print Options Dialog Box (Figure 6-4).



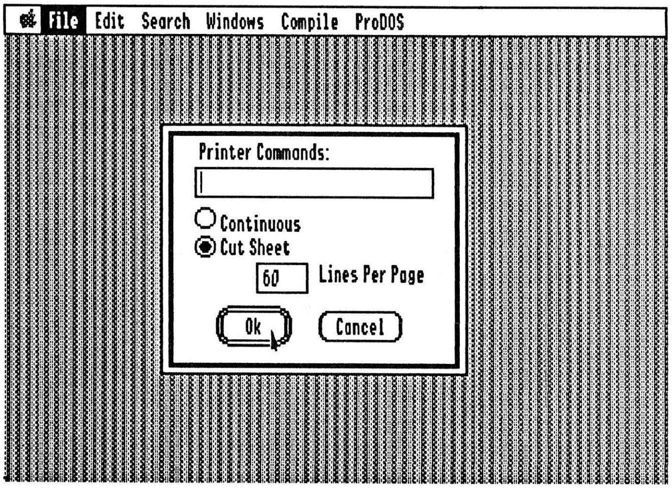
Print Options
☒ Print Title
☒ Print Date/Time
☒ Print Page Numbers

Figure 6-4
Print Options Dialog Box

When TML BASIC prints a file to the printer, it optionally prints a header across the top of every page. The header can include the name of the file (Print Title), the current date and time (Print Date/Time), and page numbers (Print Page Numbers). If an option is checked, TML BASIC prints the corresponding information in the header. If none of the options are selected, a header is not printed.

Page Setup...

This menu item displays the Page Setup Dialog Box (Figure 6-5).



Printer Commands:

( ) Continuous
[✓ Cut Sheet]

60 Lines Per Page

Figure 6-5
Page Setup Dialog Box

The Page Setup Dialog Box is used to configure the way TML BASIC prints a page. There are two options: Continuous and Cut Sheet. If Continuous is selected, a header is only printed on the first page, and no blank lines are printed at the end or beginning of a piece of paper. This option maximizes the number of lines that can be printed on a page. However, if the paper is misaligned, a line of text may print on the perforation in the paper.

If the Cut Sheet option is used, a header is printed at the top of every page, and blank lines are printed at the end and beginning of every page. When this option is selected, the number of lines per page must be set. The default setting is for standard 8 1/2 by 11 inch paper.

Finally, the Page Setup Dialog Box allows you to enter a special character sequence which represent a Printer Command. The character sequence is sent to the printer before printing every file. The Printer Command can be used to instruct a printer to use a special built in font or font size, page size, etc. In order to send a control character to the printer use the caret character (^) followed by the appropriate letter that defines the control character. For example, ^? sends an ASCII ?? (an escape character). See Appendix E for these codes.

Chooser...

The **Chooser** menu item allows you to specify to TML BASIC which of the two serial ports (printer port or modem port) your printer is connected. Your selection is saved to the TMLBASIC.OPTS file so that you do not need to specify your selection every time you use TML BASIC. If you have changed the Apple IIGS Control Panel so that either Slot 1 or Slot 2 does not use the built-in port, but rather a card in that slot, TML BASIC will obey this change so that you can use a parallel printer with an appropriate card.

Print ⌘P

The **Print** menu item causes the contents of the active window to be printed to the printer through the currently selected serial port (slot). The text is printed using the built-in font of the printer. TML BASIC does not use the Apple IIGS Print Manager for printing.

If the Option key is held down when choosing this command, TML BASIC prints the currently selected text in the active window rather than the entire contents. This is especially useful when editing large files.

Quit ⌘Q

Selecting **Quit** closes all open windows, allowing you to verify whether changes made to each window should be saved, and then exits back to the Apple IIGS Finder.

The Edit Menu

The **Edit** menu contains several useful editing commands. The menu is in the standard Apple IIGS format thus allowing it to be used with desk accessories. See Figure 6-6.

Undo ⌘Z

Undo does not support the TML BASIC editing windows and is therefore disabled. However, **Undo** is enabled whenever a *desk accessory window* is the active window so that it is available for the desk accessory.

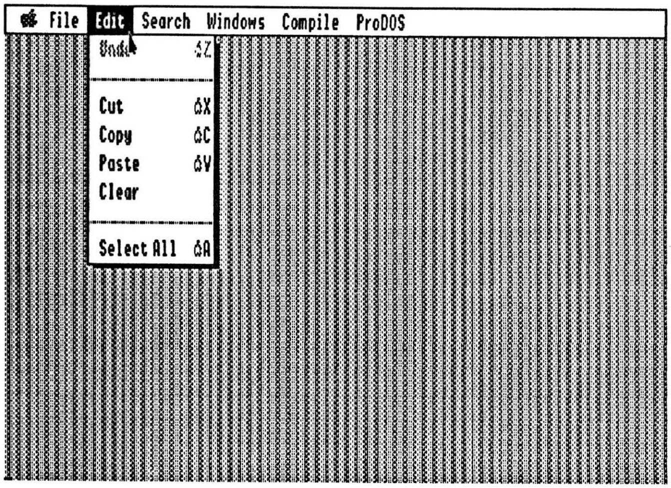


Figure 6-6
Edit Menu

Cut ⌘ X

This command cuts the currently selected text. The operation deletes the selected text from the active window and saves it into the clipboard.

Copy ⌘ C

This command copies the currently selected text, but does not delete it from the active window, and saves it into the clipboard.

Paste ⌘ V

The **Paste** menu item copies the contents of the TML BASIC clipboard into the active window at the current insertion point. If text is currently selected then it is deleted before the paste is perfomed.

Clear

The **Clear** menu item deletes the currently selected text from the active window, but does not save it into the clipboard.

Select All ⌘A

This command selects all text contained in the active window. It is a shortcut for selecting all text by moving to the beginning of the text, clicking the mouse, and then moving to the end of the text and shift-clicking.

The Search Menu

The **Search** menu contains a collection of commands which perform search and replace operations (Figure 6-7). The **Search** menu also contains two different Goto commands that move the insertion point to a specified location in your source code.

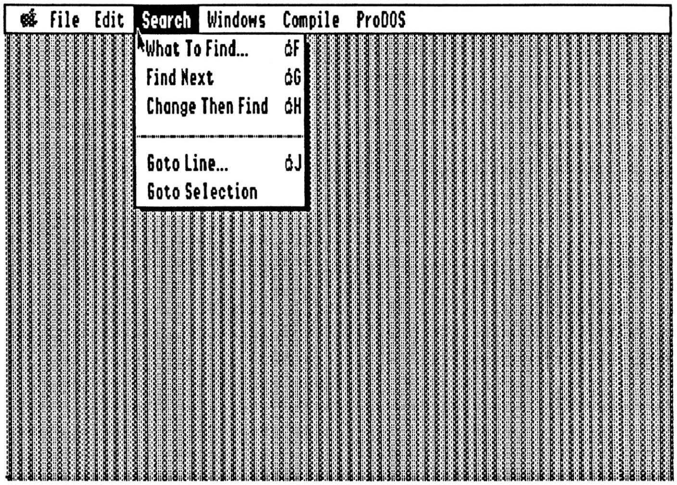


Figure 6-7
Search Menu

What to Find... ⌘F

This menu item displays the TML BASIC Find Dialog Box allowing you to specify a search string and an optional replacement string. When choosing this command, if the active window has a range of text selected which resides on a single line, then TML BASIC automatically makes this the default search string.

When selecting the Find button in this dialog, the search begins from the current insertion point (not the beginning of the file). Selecting the Change All button instructs TML BASIC to change every occurrence of the search string with the replacement string beginning from the current insertion point to the end of the file.

Find Next G

This command searches forward in the active window, from the current insertion point, for the next occurrence of the **Find What** string specified in the Find Dialog Box. Upon locating the next occurrence, the active window scrolls to display the string. If no occurrence of the string is found an error message is displayed.

Change then Find H

The current selection in the active window is replaced with the **Change To** string last specified in the Find Dialog Box. The command then searches forward in the active window for the next occurrence of the **Find What** string. If an occurrence of the string cannot be found then an error is reported.

Goto Line... J

The **Goto Line** menu item allows you to move the insertion point to the beginning of a specified line within the active window. Upon selecting this command the Goto Line Dialog Box is displayed allowing you to specify a line number you wish to be placed on. The default line number is "END" which signifies the last line of the file.

Goto Selection

This command scrolls the active window so that the insertion point (or currently selected text) is visible in the window.

The Windows Menu

The **Windows** menu provides three commands to arrange the open windows within the TML BASIC desktop. Figure 6-8 shows the contents of this menu.

Stack Windows

The **Stack Windows** menu item allows you to neatly organize your editing windows in a stack. The current editing window remains active while the others are stacked behind it with their title bars showing. This command is useful when you have moved the open windows and wish to rearrange them neatly.

Tile Windows

This menu item arranges the open windows so that none of the windows overlap. Using this window configuration allows you to see each open window's source code without moving from one window to another.

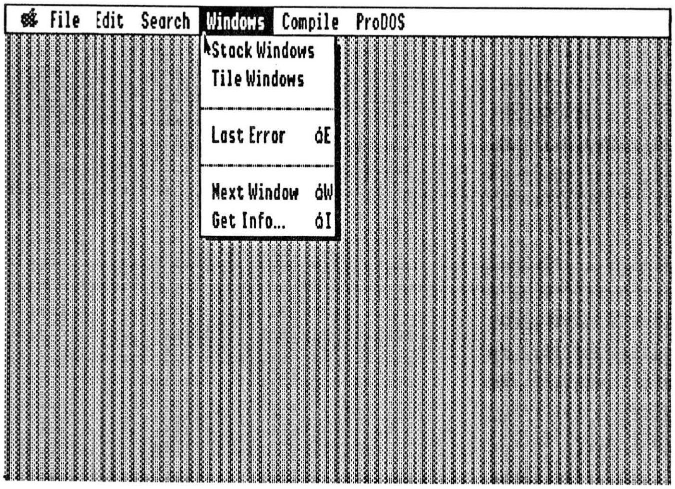


Figure 6-8
Windows Menu

Last Error ⌘E

This command displays the TML BASIC Error Dialog Box, displaying the most recently encountered error.

Next Window ⌘W

The **Next Window** menu item places the active window in back of all other open windows on the screen and brings the window directly behind the previously active window to the front. This command provides an easy method of switching between windows when it might not be possible to click on a window because it is completely covered by another window.

Get Info... ⌘I

The **Get Info** command displays a File Information Dialog Box. The dialog box displays the following information about the active editing window: the full pathname for the file associated with the editing window, its size in bytes and the number of lines.

The Compile Menu

The **Compile** menu (Figure 6-9) contains the commands which invoke the TML BASIC compiler. When invoking the compiler, the contents of the active editing window are compiled. Also included is the **Preferences...** command which allows you to customize the way TML BASIC operates.

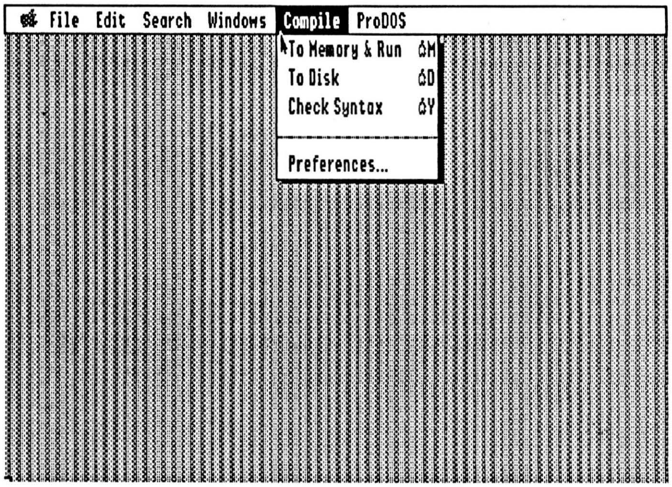


Figure 6-9
Compile Menu

To Memory & Run ⌘M

This command invokes TML BASIC to compile the source code contained in the active editing window. If the compilation completes successfully and the active window contains a program which is an application (rather than a library), the state of TML BASIC, including all open windows, is saved and control is transferred to the compiled application. Upon quitting the compiled program, you are returned to TML BASIC with all of your windows intact.

If the contents of the active window is a library rather than a program then there is no program to run and, therefore, no transfer of control out of TML BASIC. Instead, the compiled code for the unit is retained in memory so that other libraries and programs which use the unit will have access to its code.

To Disk ⌘D

The **To Disk** menu item invokes the TML BASIC compiler to compile the contents of the active window to disk creating a stand-alone ProDOS application file.

If the source code contained in the active window is a program then TML BASIC creates an *S16* (filetype $B3) application load file in the same directory as the source code. On the other hand, if the source code is a library then the library's symbol table and code are saved to a .LIB file in the same directory as the source code.

For detailed information regarding file naming conventions and other related information, see Chapter 3.

Check Syntax Y

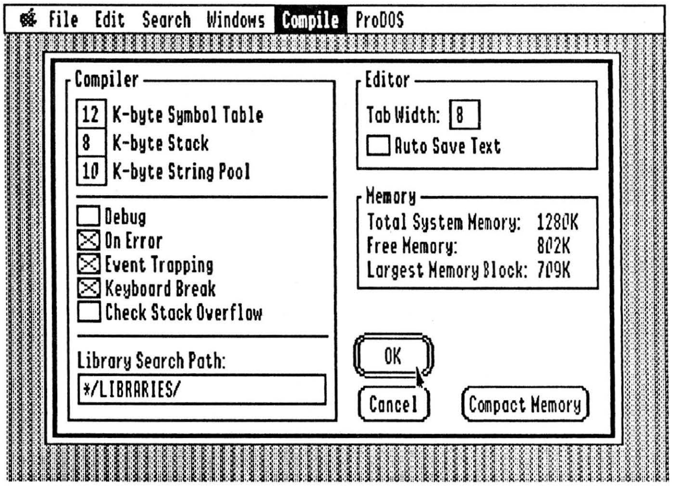
The **Check Syntax** command invokes the TML BASIC compiler only to verify that the source code contained in the active window consists of legal BASIC statements.

Preferences...

Selecting the **Preferences** command displays the Preferences Dialog Box. The Preferences Dialog Box is used to configure the TML BASIC editor and compiler to your particular needs. The information presented in the dialog is grouped into three major categories: Compiler, Editor, Memory. In addition, there are three buttons: OK, Cancel and Compact Memory. The Preferences Dialog is displayed in Figure 6-10 with its default settings. The next several paragraphs describe each component of the dialog in detail.

Before discussing each component of the Preferences Dialog, an explanation of *edit text items* and *check boxes* is in order. An edit text item is an item contained in the Preferences Dialog which requires input to determine a components value, whereas a check box acts as an on/off switch. These two mechanisms are the means by which you modify each component of the Preferences Dialog Box.

Simply position the cursor over an edit text item, click once and begin typing to enter the value for its component. Check boxes, on the other hand, are modified by positioning the cursor over the check box and clicking the cursor once to toggle between on and off states (a check representing "on").



```
Compiler
[12] K-byte Symbol Table
[8 ] K-byte Stack
[10] K-byte String Pool

[ ] Debug
[☒] On Error
[☒] Event Trapping
[☒] Keyboard Break
[ ] Check Stack Overflow

Library Search Path:
*/LIBRARIES/

Editor
Tab Width: [8]
[ ] Auto Save Text

Memory
Total System Memory:  1280K
Free Memory:           802K
Largest Memory Block:  709K
```

Figure 6-10
Preferences Dialog Box

K-byte Symbol Table This option allows you to specify the amount of memory the TML BASIC compiler should allocate for a *symbol table.* A symbol table is the data structure the compiler uses to store the declarations of labels, variables, arrays, procedures and functions. For most all programs, the default size of 12K bytes is sufficient. However, larger programs may encounter the Compiler error Symbol Table Space Exhausted (not to be confused with Out of Memory). If the compiler of a program encounters this error, then you should increase the value of this setting. 32K bytes is the largest setting allowed. This setting can also be lowered if you are running short of memory and are compiling small programs. The smallest allowable value is 2K bytes.

K-byte Stack TML BASIC programs require a data structure known as the *Runtime Stack.* The runtime stack is used to implement certain TML BASIC statements including PROC, GOSUB and LOCAL. The default value of a 8K byte stack should be sufficient for most TML BASIC programs. This value can be changed from 1K to 32K bytes. See Chapter 8 for more information about the Runtime Stack.

The Stack size may also be changed by using the $StackSize metastatement. See Appendix B.

K-byte String Pool

The *String Pool* is where TML BASIC stores all values of string variables for a program. If you assign a string constant to a string variable, that string constant is copied to the string pool. If a program is running out of string space this value should be increased. The maximum size for the string pool is 64K bytes, the minimum is 1K bytes. For more information about strings, string data and the string pool see Chapter 7.

The String Pool size may also be changed by using the $StringPoolSize metastatement. See Appendix B.

Debug

When the Debug option is turned on, the TML BASIC compiler generates code to support the TML BASIC debugger. The generated code checks for all runtime errors such as Overflow Error, Illegal Quantity Error, etc. It also generates a special data structure called the *line number table* so that the TML BASIC debugger can determine in what line of source code the runtime occurred. If this option is turned off, then all runtime errors will go undetected. The runtime errors are listed in Appendix A.

The Debug option makes programs larger and slower to execute. The option should be turned off when a program is known to be correct, and no longer requires the debug code.

This option may also be turned off and on using the $Debug metastatement. See Appendix B.

On Error

The On Error option is used to indicate to the TML BASIC compiler that a program contains the ON ERR statement along with the statements RESUME and/or RESUME NEXT. These statements require that the *line number table* be generated so that TML BASIC can determine on which line to resume or resume next after an error has been handled by an ON ERR statement list.

If your programs do not contain these statements then it is best to turn this option off since it will decrease the size of your applications.

This option may also be turned off and on using the $OnError metastatement. See Appendix B.

Event Trapping

This option must be turned on when a program contains statements requiring event trapping. These statements are the ON KBD and ON TIMER. When these statements are used, TML BASIC must generate code between each statement to check for the occurrence of a keyboard or timer event. This option should only be turned on when a program contains these statements since the code necessary to check for these events makes a program larger and slower to execute.

This option may also be turned off and on using the $EventTrapping metastatement. See Appendix B.

Keyboard Break

The Keyboard Break option is used to implement the ON BREAK statement. It is also required to allow a program to be aborted by typing a control-C.

If this option is turned on, TML BASIC generates code between each statement to check if the control-C character has been typed. If this option is turned off, it is impossible to abort the execution of a TML BASIC program. The only way to do so is to reset the Apple IIGS. If you do not intend to abort the execution of your programs and do not use the ON BREAK statement then you should turn this option off so that your programs will be smaller and faster.

This option may also be turned off and on using the $KeyboardBreak metastatement. See Appendix B.

Check Stack Overflow

This option is used to instruct the TML BASIC compiler to generate code for each procedure and function's entry code to check to make sure that there is sufficient space in the runtime stack to call the procedure and allocate its local variables. If there is insufficient space, the runtime error Stack Overflow occurs (If the Debug option is turned on then the TML BASIC debugger is capable of showing you what procedure or function caused the stack overflow).

Most programs never need more stack space than the default 8K bytes, thus this option is turned off by default. However, if your program is behaving very strangely, it may be that its stack is growing too large and destroying memory. Turning this option on will determine if your program does indeed have this problem. If it does, you should increase the allocated stack space for the program.

This option may also be turned off and on using the $CheckStack metastatement. See Appendix B.

Library Search Path The pathname specified here is where the TML BASIC compiler searches for library files which have been specified in a LIBRARY statement. The default value for this option is */LIBRARIES/ which specifies the LIBRARIES folder on the boot disk. This is the folder which contains all of the TML BASIC predefined libraries for accessing the Apple IIGS Toolbox. Recall that TML BASIC first searches in the current source code folder first for a library file and then the path specified by this option.

Tab Width This option is used by the Editor to determine how many spaces wide a tab stop is. The default value for this option is 4 spaces. Any value between 2 and 10 is legal.

Auto Save Text The Auto Save option allows you to specify whether or not changes to any of the editing windows should be automatically saved before TML BASIC transfers control to a compiled to memory application. You should select this option if your program is in the early stages of development and might cause the Apple IIGS to crash when run. If this option is on you will never lose any editing changes you have made, but not explicitly saved, however, it does slow down the compile cycle since it must write to the disk.

Total System Memory Obviously, this value can not be changed while TML BASIC is running. The total system memory is displayed for informational purposes only. The value represents, in kilobytes, the amount of RAM memory installed in your machine.

Free Memory This number indicates how much memory is currently available. This number is important because, it reflects whether or not TML BASIC has enough memory to open a new program file, compile a program to memory, etc. Because TML BASIC retains various pieces of information in memory, this number can sometimes be increased by selecting the Compact Memory button described below.

Largest Memory Block This value indicates the largest block of memory available for use by TML BASIC.

OK Button Clicking the mouse in this button (or pressing the Return key) indicates that you want TML BASIC to accept all the changes to the Preferences dialog you have made. After choosing this button, the dialog disappears and TML BASIC updates all options and settings.

Cancel Button Clicking the mouse in this button causes TML BASIC to remove the dialog, and to ignore any changes made to the options and settings and to leave them as they were before the dialog was opened.

Compact Memory This button is used to release all memory associated with programs and libraries that have been compiled to memory or loaded to memory by a LIBRARY statement. Selecting this button will usually adjust the Free Memory and Largest Memory Block values. Clicking the mouse in this button does not cause the dialog box to close.

The ProDOS Menu

The **ProDOS** menu provides access to three ProDOS16 commands (Figure 6-11).

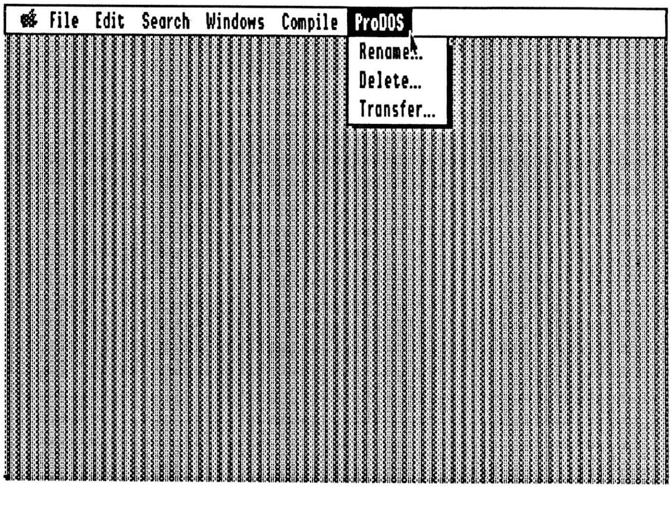


Figure 6-11
ProDOS Menu

Rename...

The **Rename** command displays the Rename File Dialog Box allowing you to choose the file you would like to rename. After selecting a file, you are prompted to provide the new filename. The new filename must be a legal ProDOS16 filename otherwise an error results.

Delete...

The **Delete** command displays the Delete File Dialog Box allowing you to choose a file you would like to delete. After selecting a file, you are prompted to confirm that in fact you would like to delete the file. If you confirm that the file should be deleted, TML BASIC will permanently delete the file from the disk.

Transfer...

The **Transfer** command displays the Transfer Dialog Box allowing you to choose an application you would like to transfer control. Upon selecting an application, TML BASIC asks you to save any changed files and then automatically quits and invokes the specified application without returning to the Apple IIGS Finder. The only way to return to TML BASIC is to launch it from the Finder again.

Command-Keys versus the Mouse

As discussed earlier in this manual, TML BASIC provides an alternative to positioning the cursor over a menu, clicking the mouse button once and holding it down as you drag and release the cursor over a menu item.

Many menu commands can be invoked by typing the menu's *command-key equivalent*. Command-keys are quite useful for invoking a menu command when your hand is on the keyboard, and not the mouse. However, a few menu commands (generally the less used items) do not have command-keys.

Additionally, when the active window contains a desk accessory, no command keys are available except for the **Edit** menu commands. The reason for this is that the Apple IIGS Desk Manager automatically captures command keys for desk accessories before TML BASIC even has a chance to see them. In this case, you will either have to use the mouse to select menu commands or make a different editing window the topmost window.

One final note. If you are using an upgraded Apple IIe as an Apple IIGS, and do not have a new IIGS keyboard, the Apple key is the same as the Open-Apple key immediately to the left of the space bar on the Apple IIe keyboard. The Closed-Apple key to the right of the space bar is equivalent to the Apple IIGS Option key.

Part II

TML BASIC Language Reference

Chapter 7

Language Elements

This and the following three chapters represent a technical discussion of the TML BASIC language. If you are a beginning programmer, and require a less technical introduction to the TML BASIC language, you should read Part I of this manual with emphasis on Chapter 5. For those programmers interested in programming the Apple IIGS Toolbox with TML BASIC, Part III of the manual provides the information you will need.

Source Code Structure

The fundamental component of every TML BASIC program is the *statement*. TML BASIC programs consist of zero or more lines of statements. Of course, a program with zero statements is not very useful. Each line of TML BASIC source code is of the form:

[*label* :] *statement* { : *statement* } [' *comment*]

or,

$*metastatement*

Before proceeding any further, the notation used in the above example to describe TML BASIC statements and lines deserves some explanation. The notation consists of four parts: BASIC reserved words or special characters, brackets ([ ]), braces ({ }), and italicized words.

BASIC reserved words and special characters appear in normal type and in all capital letters (see Tables 7-1 and 7-2). When these words or characters appear in our notation, they must be used exactly as shown. The brackets are used to indicate that all the symbols which appear between matching left and right brackets may *optionally* appear in the statement or line being described. The braces are used to indicate that all the symbols which appear between matching left and right braces may appear *zero or more* times. That is, the symbols may not appear at all, one time, two times, three times, etc. Finally, italicized words are used to denote a sequence of one or more legal TML BASIC language elements that must obey certain rules. When an italicized word appears, it is usually followed by a sentence or more which describes what the word represents. An attempt is also made for the word itself to communicate its meaning.

Now, lets examine the structure of a TML BASIC source code line as defined by our notation.

A source code line may optionally begin with a *label*. A label is a sequence of one or more alphanumeric characters that begins with a letter and does not spell any of the TML BASIC reserved words. A label may also contain the period (.) character. If a label is used, it must always be followed by a colon regardless if anything else appears on the line. However, when a label is referenced in a statement (for example, *GOTO myLab*) the colon should not appear.

The following are legal TML BASIC labels:

```
    HandleError:
    Lab183:
    SCREENUPDATE:
    Label.with.periods:
```

Case is insignificant in the spelling of a label. Thus, the following represent the same TML BASIC label:

```
    MYLAB:
    MyLab:
    mylab:
```

Note that TML BASIC statements *do not* begin with line numbers. In fact, line numbers do not exist anywhere in a TML BASIC program.

Statements are the fundamental component of TML BASIC programs. There is an extensive collection of statements available in TML BASIC (see Chapter 10). A line may contain one, or several statements, each separated by a colon. The only restriction on the number of statements that may appear on a single line is TML BASIC's restriction that source code lines are limited to 255 characters. Of course, it is generally good practice to limit the length of a line to the width of an editing window. Programs also print better when the length of a line is kept at a reasonable size. Unlike most BASIC interpreters, there is no penalty in size or speed of a program for more lines. Additionally, because the TML BASIC debugger only determines what *line* an error occurred in, the debugger is a more useful tool when only one statement appears on a line.

Note that it is legal to have an *empty statement*. Thus, it is possible to have blank lines in your program. Blank lines are good for organizing your program's code so that different sections of the program are visually separated.

Finally, a *comment* may be added to the end of any line. A comment begins with the single quote (') character followed by any descriptive text about the program.

The comment continues to the end of the line. Comments may be used in place of REM statements unless the comment appears after a DATA statement. In this case, TML BASIC interprets the single quote as part of a string in the DATA statement. Unlike, the REM statement, a colon is not needed to separate the comment from a preceeding statement. For example,

```
    Interest = Principle * Rate       ' Calculate the interest due
    Interest = Principle * Rate       : REM Calculate the interest due
```

MetaStatements are special commands to the TML BASIC compiler that direct it to behave in certain ways. A line which contains a metacommand must begin with the dollar character ($) followed by the name of the metacommand and any parameters. Only one metacommand may appear on a line. TML BASIC provides metacommands for most of the compiler options which appear in the Preferences dialog box. A complete list of the TML BASIC metacommands and their use is given in Appendix B.

Programs

The TML BASIC compiler recognizes two types of source code structures – *Programs* and *Libraries*. A program is a collection of statements that perform some action. When a program is compiled, TML BASIC creates a complete and stand-alone application which can either be run immediately from within the TML BASIC environment using the **To Memory & Run** compile option or can create a ProDOS 16 application file using the **To Disk** compile option. A compiled ProDOS 16 application file can then be run from the GS Finder by double-clicking on its icon.

Libraries

Libraries, on the other hand, cannot be run. A library is considered a *repository* for pieces of code. When TML BASIC compiles a library, it saves the compiled code so that other programs and libraries can use its code just as if its source lines were textually included in the program that uses it. When a library is compiled to disk, its code is saved in a special file ending with the suffix .LIB. The source code for a library is different from a program in that it must have a specific structure. That is, it must begin with the statement DEF LIBRARY and end with the statement END LIBRARY. For a complete discussion of libraries see Chapter 8.

See Chapter 3, "Creating a Stand-Alone Application", for a discussion of the TML BASIC file naming conventions for compiled programs and libraries.

TML BASIC Character Set

TML BASIC character set consists of alphabetic characters, numeric characters, and several special characters. The alphabetic characters are uppercase letters (A-Z) and the lowercase letters (a-z). The numeric characters are the digits 0-9. The following table lists the special characters recognized by TML BASIC with a description of their usage.

Table 7-1
TML BASIC Special Characters

Symbol	Description	Function
!	Exclamation point	Structure array type character
"	Double quote	String constant delimiter
#	Number sign	Double-precision real type character
$	Dollar sign	String type character, metastatement prefix
%	Percent sign	Integer type character
&	Ampersand	Long integer type character
'	Single quote	Comment delimiter
( )	Parentheses	Parameter lists, array indexing, expression precedence
*	Asterick	Multiplication operator
+	Plus sign	Addition operator, string concatenation operator
,	Comma	Delimiter
-	Minus sign	Subtraction operator, negation operator
.	Period	Used to form variable, array, procedure, function and label names
/	Slash	Division operator
:	Colon	Statement delimiter
;	Semicolon	Delimiter
<	Less than	Relational operator
=	Equal sign	Assignment operator, relational operator
>	Greater than	Relational operator
@	At symbol	Double integer type character
^	Caret	Exponentiation operator
_	Underscore	Alternate for the CALL statement

These are the only characters that may appear in a TML BASIC program. The only exception is that comments and string constants may contain any character.

Reserved Words

TML BASIC reserves several words for special meaning, typically for a function or statement name. Reserved words cannot be used for label, variable, array, or procedure and function names. Attempting to use a reserved word as an identifier will cause TML BASIC to signal a syntax error in your program. The following table summarizes the reserved words in TML BASIC.

Table 7-2
TML BASIC Reserved Words

ABS	AND	ANU	APPEND
AS	ASC	ASSIGN	ATN
AUXID	BDF	BREAK	BTN
CALL	CAT	CATALOG	CHAIN
CHR	CLEAR	CLOSE	COMPI
CONV	COS	CREATE	DATA
DATE	DEF	DELETE	DIM
DIV	DO	DYNAMIC	ELSE
ELSEIF	END	EOF	EOFMARK
ERASE	ERR	ERROR	EVENTDEF
EXCEPTION	EXEVENT	EXFN	EXP
EXP1	EXP2	FILE	FILTYP
FIX	FN	FOR	FRE
FREMEM	GET	GOSUB	GOTO
GRAF	HEX	HOME	HPOS
IF	IMAGE	INPUT	INSTR
INT	INVERSE	JOYX	JOYY
KBD	LEFT	LEN	LET
LIBRARY	LOCAL	LOCATE	LOCK
LOG	LOGB	LOG1	LOG2
MENUDEF	MID	MOD	NEGATE
NEXT	NORMAL	NOT	OFF
ON	OPEN	OR	OUTPUT
PDL	PDL9	PEEK	PFX
PI	POKE	POP	PREFIX
PRINT	PROC	PUT	R.STACK
RANDOMIZE	READ	REC	REM
REMDR	RENAME	REP	RESTORE
RESUME	RETURN	RIGHT	RND
ROUND	RUN	SCALB	SCALE
SECONDS	SET	SGN	SHOWDIGITS
SIN	SPACE	SPC	SQR
SRC	STEP	STOP	STR
SUB	SWAP	TAB	TAN
TASKPOLL	TASKREC	TEN	TEXT
TEXTPORT	THEN	TIME	TIMER
TO	TXT	TYP	UBOUND
UCASE	UIR	UNLOCK	UNTIL
UPDATE	USING	VAL	VAR
VARPTR	VPOS	WHILE	WRITE
XOR			

Numbers in TML BASIC

Numbers are one of the most important components of any program. They allow a program to count, compute, calculate and otherwise do its job. Traditionally, BASIC implementations have provided only a single type of number for programs to do its job. Because of this, programs have had little concern for such matters as the speed, precision and memory requirements of numbers – there was only one type of number, and there was no choice to be made.

However, TML BASIC offers several "types" of numbers: *integers, double integers, long integers, single-precision real* and *double-precision real.* Thus, programs that require space efficiency or faster calculations can use the smaller, faster numbers, while those that require a high degree of accuracy can use the slower, but more precise numbers.

Integers

The smallest, fastest data type in TML BASIC is the *integer* type. An integer is a number with no decimal point and is in the range -32,768 to 32,767. The reason for this range of values is that integers are stored as 16-bit (two bytes) signed values. One bit is used to indicate the sign of the number and 15 bits are used to represent the magnitude: 2^{15} is 32,768.

While integers are somewhat constrained in the range of values they can represent, they make up for this with their speed. The Apple IIGS is most efficient when handling *integer* numbers. Programs should use integers in FOR...NEXT loops, as counters, etc. in order to produce the most efficient code possible.

Double Integers

Double integers behave just like normal integers with the exception that double integers provide significantly more precision than integers. Double integer values range from -2,147,483,648 to 2,147,483,647. Of course, to achieve this much precision, twice as much storage is required than integers – four bytes. Whenever integers will not suffice, try to use double integers next, since these numbers are the second fastest and smallest in TML BASIC.

Long Integers

Again, *long integers* behave just like integers and double integers, except that they provide an incredible range of values: -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807. Long integers require eight bytes of storage (64 bits). Long integers provide the greatest range of values among the integer types, however, they are the slowest integer type for performing computations.

Many types of financial programming can take advantage of this data type. In order to avoid any round-off errors, calculations can be performed in pennies using long integers, then scaled when output.

Single-Precision Reals

Single-precision reals are the smallest of the real numbers. A single-precision value is a number that may contain a decimal point and an exponent in the approximate range -3.4E38 to 3.4E38 with up to 7-8 significant digits. That means you can have a number as small as 0.00000000000000000000000000000000000001 or as large as 100,000,000,000,000,000,000,000,000,000,000,000,000, but of course only 7 to 8 digits are significant. Single-precision reals occupy 4 bytes of memory.

While the single-precision reals represent a very large *range* of numbers, it does have a somewhat limited number of significant digits. When a large amount of precision is necessary, the long integers or double-precision reals may be more appropriate.

Double-Precision Reals

Double-precision reals are the largest of the real numbers in TML BASIC. They require twice as much space as the single-precision reals (8 bytes), but they have a much greater range and precision. A double-precision value is a number in the approximate range -1.7E308 to 1.7E308 with up to 15-16 significant digits.

Double-precision real numbers should be able to handle any computational need in your program. However, use it with care in large arrays, etc. because of the large amount of storage it requires.

Extended-Precision Reals

There is actually one more type of number in TML BASIC called the *extended-precision real.* Extended-precision is an internal number type that is used by TML BASIC to perform all of its real number calculations. That is, whenever a program performs an arithmetic, relational or logical operation (discussed later in this chapter), TML BASIC internally uses extended-precision numbers. TML BASIC will automatically convert single-precision and double-precision numbers to extended-precision before performing an operation and then convert the extended-precision result back to the appropriate type. This technique allows TML BASIC to avoid round-off errors in its calculations, thus providing your program with the most accurate result possible.

The extended-precision number requires 10 bytes (80 bits) of storage and can represent values in the approximate range -1.1E4932 to 1.1E4932 with 19 to 20 significant digits.

Understanding the implications of TML BASIC's technique for performing real number calculations is important to writing successful programs. While this technique offers your programs greater accuracy for its calculations, there are some side effects you should be aware of when writing programs. Since the extended precision number provides a greater number of significant digits, some information may be lost when converting the result of an expression to a single- or double-precision number. Thus, you might be surprised with the following code's output:

```
aDblReal# = 1.0/3.0
IF aDblReal# = 1.0/3.0 THEN PRINT "Everything is working fine"
ELSE                        PRINT "Something strange is going on"
```

When this program is executed, the output is "Something strange is going on". The reason for this is that the extended precision result of the operation 1.0/3.0 contains significant digits beyond the storage capacity of the double-precision reals. Thus, when the double-precision variable, aDblReal#, is assigned the result of this calculation it is no longer *exactly* equal to the representation of this number in extended-precision. When the test for equality is done, extended-precision is used, and of course the values are not equal.

To solve this problem, a program should ensure the representation of both operands of a relational operator are in the same precision. This can be done by making sure both operands are simple variables of the same type, or by using one of the CONV functions. For example, the following variation of the program produces the expected result.

```
aDblReal# = 1.0/3.0
IF aDblReal# = CONV#(1.0/3.0) THEN PRINT "Everything is working fine"
ELSE                               PRINT "Something strange is going on"
```

Because of the way TML BASIC uses extended-precision numbers, it may sometimes be advantageous to perform a calculation in one large expression rather than several smaller ones which store temporary results in a real type with less precision.

Note, that all TML BASIC built-in numeric functions which return a real number, actually return an extended-precision result.

The Standard Apple Numeric Environment (SANE)

The Standard Apple Numeric Environment, also called SANE, is the floating point engine used by TML BASIC for all of its real number operations and functions. The SANE implementation is based upon the IEEE Standard 754 for Binary Floating-Point Arithmetic and provides for an extensive collection of numeric operations, not all of which are available in TML BASIC.

SANE is a standard toolset in the ROM of every Apple IIGS. For more information about SANE, see the *Apple Numerics Manual.*

Strings in TML BASIC

The *string type* is the only other type of data available in TML BASIC besides the five number types. A string is a sequence of characters along with a count that indicates how many characters are in the string. In TML BASIC, a string may contain zero characters up to a maximum of 255 characters. If a string contains zero characters, it is said to be a *null-string*.

In TML BASIC, all strings are stored in a special area of memory called the *String Pool.* When a sequence of characters is assigned to a string variable, the characters are copied to the string pool. These characters are called the *string data.* A string variable requires only two bytes of memory which is an offset into the string pool to where its string data is stored. The string data consists of a sequence of bytes containing the characters in the string. There is also an additional byte of memory which stores the number of characters in the string.

Strings are extremely powerful in TML BASIC. They can be converted to numbers and back to strings again, and can be manipulated with a large assortment of predefined string functions.

Constants

Constants are predefined values that do not change during the execution of a program. There are two types of constants in TML BASIC: numeric and string.

Numeric Constants

Numeric constants are positive or negative numbers. There are three classes of numeric constants in TML BASIC: integer, fixed-point and floating-point.

An integer constant contains only the digits 0-9 with an optional sign prefix (+ or -). If the constant is in the range -32,768 to 32,767, the constant is treated as an integer. Otherwise, the constant is treated as a double integer and must be in the range -2,147,483,648 to 2,147,483,647. Integer constants outside of this range are errors. Examples of integer constants:

```
1
2569
721039
```

A *fixed-point constant* contains the digits 0-9 with an optional sign prefix and a decimal point. All fixed point constants are treated as extended precision reals. Examples of fixed-point constants follow:

```
1.0
34.9
238540923423.482
```

Finally, a *floating-point constant* is represented in exponential form. A floating-point constant consists of a fixed-point number called the mantissa followed by the letter E or e and an optionally signed integer exponent. The exponent is a power of ten, by which the mantissa is multiplied to obtain the value of the floating-point number. All floating-point constants are treated as extended precision reals. Examples of floating-point constants follow:

```
1.0e0
349.2001E-23
9.98765e+78
```

String Constants

String constants are simply a sequence of zero to 255 printable ASCII characters enclosed by double quotes. For example:

```
"Apple IIgs"
"123.45"
""
```

It is not possible to create a string constant which contains the double quote character since the double quote is used to delimit the string constant. Instead, the CHR$ function can be used to create a single character string which is the double quote. For an example of how this is done see the *CHR$ Function* in Chapter 10. When the two double quotes appear next to each other, no characters are in the string constant. This is called a *null-string*.

TML BASIC provides one special exception to the above rule. The non-printable *null* character whose ASCII value is zero (0) may be represented in a string constant with the backslash character (\) followed by the zero character (0). For example:

```
"Hello\0"
```

This string constant consists of six characters: H, e, l, l, o, and the *null* character.

This exception is provided for easier programming of the Apple IIGS Menu Manager for creating programs which use menus. For more information on how to program with the Menu Manager and the other Apple IIGS Toolbox libraries see Chapters 11 through 13.

Variables

A *variable* is a named entity which represents a numeric or string value. Unlike a constant, the value of a variable can change during the execution of a program. The name of a variable must begin with a letter followed by any number of letters and digits that do not spell any of the TML BASIC reserved words. A variable name may also contain the period (.) character. TML BASIC places no restriction on the length of a name except that it must fit on a line (255 characters). The length of a name has no effect on the size or execution speed of a program, so you should always use descriptive and meaningful names for variables.

TML BASIC supports six different variable types. The last character of a variable name determines its type. Table 7-3 lists the legal type characters for TML BASIC. Note that if no type character appears after a variable name, it is treated as a single-precision real.

Table 7-3
TML BASIC Type Characters

%	Integer	2 bytes
@	Double Integer	4 bytes
&	Long Integer	8 bytes
	Single-precision real	4 bytes
#	Double-precision real	8 bytes
$	String	2 bytes for string variable
		1 byte for each character in the string
		1 byte to store the length of the string

Variables are created when they are first used in a program. When TML BASIC sees a variable name in a statement, it first checks to see if a variable with the same name has already been used in the source code above the current line. If it has, TML BASIC knows where in memory to read or store the value of the variable. If not, TML BASIC automatically enters the name into its symbol table and allocates storage for the variable.

Note that X%, X@, X&, X, X# and X$ are different variable names.

TML BASIC initializes all numeric variables with the value zero and all string variables with the null string.

Reserved Variables

TML BASIC provides a collection of *reserved variables*. A reserved variable is special in the sense that in most cases, it does not have memory allocated for it in the normal way. But instead is associated with a special feature of the Apple IIGS. For example, the PI reserved variable contains the value of π, but that value is actually obtained from SANE.

Programs can only read the value of most reserved variables. However, some reserved variables may be given new values. These are called modifiable reserved variables. The HPOS reserved variable is a good example of a modifiable reserved variable. HPOS contains the value of the horizontal location of the text screen cursor. When you read this variable, the text screen firmware is actually read to obtain this value. Since this is a modifiable reserved variable, a new value may be assigned to HPOS. In this case, the text screen firmware is instructed to change the horizontal location of the cursor to the new value.

The name of a reserved variable is one of the TML BASIC reserved words. All of the reserved variables are discussed in Chapter 10.

Arrays

An *array* is a collection of values of the same type referred to by the same variable name. The individual values of an array are called elements. Array elements are also variables, and can be used anywhere a simple variable can be used. As with variables, the last character of the array name defines the type of the array elements. The process of declaring the name, element type and setting the number of elements in the array is known as *dimensioning* the array.

Array elements can be any of the simple variable types: integer, double integer, long integer, single-precision real, double-precision real and string. In addition, TML BASIC supports a special array type called the *structure array*. The structure array element is a byte size integer value in the range 0 through 255. The type character for a structure array is the exclamation point (!). Structure arrays are described in more detail below.

Dimensioning Arrays

The DIM statement is used to declare the name, element type, number of dimensions and size of each dimension of an array. For example:

```
    DIM Sales%(11)
```

creates a one-dimensional array variable *Sales%*, consisting of 12 integer elements, numbered 0 through 11. Note that because the first numbered element is 0, the largest numbered element is one less than the number of elements in the array dimension. The array variable *Sales%* is distinct from the simple integer variable *Sales%*.

Arrays can have one or more *dimensions,* up to a maximum of eight. A one-dimensional array such as *Sales%* is a simple list of values. A two-dimensional array is a matrix of values with rows and columns of information. Multi-dimensional arrays are also possible, but do not have a real world analog.

```
DIM array1(4)               'One-dimensional array
DIM array2(12,8)            'Two-dimensional array
DIM array3(39,3,5)          'Three-dimensional array
```

The maximum number of elements per dimension is 32,768. The maximum total size of a single array is 64K bytes. You may have as many 64K byte arrays as available memory permits. The following table reviews the memory requirements for each element type and the total number of elements possible in a 64K array.

Table 7-4
TML BASIC Array Element Sizes

%	Integer	2 bytes	32,768 elements per 64K
@	Double Integer	4 bytes	16,384 elements per 64K
&	Long Integer	8 bytes	8,192 elements per 64K
	Single-precision real	4 bytes	16,384 elements per 64K
#	Double-precision real	8 bytes	8,192 elements per 64K
$	String	2 bytes	32,768 elements per 64K
!	Structure	1 byte	65,536 elements per 64K

When a TML BASIC program begins execution, all array elements are initialized to zero except for string arrays which are initialized to the null string.

It is possible to reference an element of an array variable which has not been declared with a DIM statement. If the reference appears on the left side of a LET statement (assignment), TML BASIC automatically declares the array variable with the same number of dimensions as are referenced. Each dimension contains 11 elements. TML BASIC declares the array so that storage exists for the value to be assigned. For example, if the statement

```
Count@(2,3) = 55
```

is executed without having first DIMensioned the array variable *Count@*, therefore TML BASIC automatically defines the array variable just as if the statement

```
    DIM Count@(10,10)
```

had preceeded the assignment statement. This is called *implicit DIMensioning*. Every element of the array *Count@* is automatically given an initial value of zero just as arrays normally declared with the DIM statement.

If an undeclared array is referenced anywhere besides the left side of a LET statement, TML BASIC *does not* automatically declare the array. Instead, a dummy zero value is returned (or null string). This is unlike most other BASIC implementations which will also implicitly declare an array any time it is referenced. Thus, if the statement

```
    PRINT Count@(4,2)
```

is executed before the array *Count@* is dimensioned, a zero is displayed.

It is generally good practice to declare arrays used in a program with the DIM statement rather than allowing TML BASIC to automatically declare them.

Dynamic Allocation

Arrays declared using the DIM statement or those implicitly declared by TML BASIC are known as *Static Dimensioned* arrays. These arrays have a fixed size which is determined by the number and size of elements in the array. TML BASIC allocates the storage for these arrays before a program begins execution, and their size may not change.

A static array must be declared with constant values for the sizes of its dimensions. If an expression or variable is used to dimension a static array, TML BASIC gives the error "Static arrays must have constant dimensions". For example, the following are illegal declarations using the DIM statement.

```
    DIM Sales%(numMonths%)
    DIM Count@(eltsNeeded%+3)
```

Further, a static dimensioned array may not be declared more than once in a program. If a program attempts to declare a static array more than once, TML BASIC gives the error "Duplicate declaration of a static array". For example, the following is not legal in TML BASIC:

```
    DIM Sales%(10)
    DIM Sales%(20)
```

Static dimensioned arrays are very efficient in TML BASIC, however, some programs require the ability to dimension an array dynamically at execution time. This is done with the DIM DYNAMIC statement in TML BASIC. The DIM DYNAMIC statement allows the program to use a variable or an expression as the number of elements in an array. For example, a program might allow a user to enter a variable number of *sales figures* into an array.

```
INPUT "How many sales figures? "; numSales%
DIM DYNAMIC Sales%(numSales%)
FOR i% = 1 to numSales%
   INPUT "Enter a sales figure: "; Sales%(i%)
NEXT i%
```

Of course, it is still possible to declare arrays with static dimension values. For example, the following statement creates a dynamic array variable with 30 elements.

```
DIM DYNAMIC Sales%(29)
```

Unlike static dimensioned arrays, dynamic dimensioned arrays can be dimensioned more than once in a program. This allows a program to change the size of an array during its execution depending upon the needs of the program. To deallocate all of the space used by a dynamic array, use the ERASE statement.

Static dimensioned arrays allow TML BASIC to generate code which is faster and smaller than code for dynamic arrays. A program should only use dynamic arrays when absolutely necessary.

Evaluation of the DIM Statement

The DIM statement works differently in TML BASIC than BASIC interpreters like GS BASIC and AppleSoft BASIC. Unlike interpreters, the TML BASIC compiler processes the DIM statement when a program is compiled and not when it is executed. Thus, TML BASIC determines the number of dimensions and elements of an array, and allocates memory for the array before the program is executed.

Because TML BASIC processes a program in its textual order, rather than its execution order, you must be careful how you use the DIM statement. The following example would execute without any error in GS BASIC, but TML BASIC would give an error for the statement marked (b). The reason for this is that TML BASIC will process the statement marked (a) before the statements below it. Since this statement references the array variable *anArr%* which has not yet been declared, TML BASIC implicitly declares the array with 11 elements. Thus, when the statement marked by (b) is processed, TML BASIC gives the error "Duplicate declaration of a static array".

```
    GOTO doDIM                          'Go dimension the array

    doLET:
(a)   LET anArr%(9) = 99                'Assign the value 99 into the 9th element
      GOTO Continue                     'Continue execution

    doDIM:
(b)   DIM anArr%(29)                    'DIMension the array with 30 elements
      GOTO doLET                        'Go assign a value

    Continue:                           'Continue program execution
```

Another example illustrating this difference between TML BASIC and BASIC interpreters is shown below.

```
    IF Flag% = 0 THEN DIM anArr%(10)
    ELSE              DIM anArr%(250)
```

Again, TML BASIC would give the error "Duplicate declaration of a static array" since it ignores the fact that during execution, only one of the DIM statements is executed.

Arrays declared using the DIM DYNAMIC statement, however, do not behave in this manner. Since the DIM DYNAMIC statement is evaluated during program execution, an array may be declared and re-declared as many times as required by the program. For example:

```
    IF Flag% = 0 THEN DIM DYNAMIC anArr%(10)
    ELSE              DIM DYNAMIC anArr%(250)
```

is legal in TML BASIC. However, it is not possible to re-declare an array using DIM DYNAMIC after it has been declared with the DIM statement.

Subscripts

Individual elements of an array variable are selected using ***subscripts*** (integer expressions within parentheses to the right of an array variable's name). For example, *Sales%(3)* references the fourth element of the *Sales%* array variable. It is not legal to use a subsrcipt value for an element of the array which does not exist. For example, the statements

```
    DIM Sales%(11)
    Sales%(20) = 44
```

will cause a runtime error because the *Sales%* array does not have an element whose subscript is 20.

When referencing an array, you must also provide a subscript for each dimension of the array. If an array has three dimensions, you must provide three subscript values when referencing the array. If you provide too few or too many subscripts, TML BASIC gives the error "Array subscript error".

Structure Arrays

TML BASIC provides a special type of array called the *structure* array for manipulating bytes of memory. The type character for a structure is the exclamation point (!). Structures may only be declared with the DIM statement and are not allowed as simple variables. The elements of a structure array are bytes of memory which are treated as unsigned integers in the range 0 through 255.

Individual structure array elements may be referenced just like normal arrays. The value of a structure element is automatically converted to an integer before it is used in an expression. TML BASIC also provides the VAR and SET statements for reading or writing successive bytes of a structure. See Chapter 10 for a discussion of these statements.

Structures are typically used for representing data structures to be used with the Apple IIGS Toolbox. See Chapter 13 and Appendix C for examples of how structures are used with the Toolbox.

Expressions

An *expression* represents a value. An expression consists of a collection of *operands* combined together by *operators* to produce a value when the expression is evaluated. Operators are special symbols representing a particular operation to perform. Operands are the constants, variables and function calls that operators work on. In TML BASIC there are two types of expressions – string and numeric.

String expressions consist of string constants, string variables and string functions, optionally combined with the string concatenation operator (+). String expressions evaluate to a string; that is, a sequence of ASCII characters with a known fixed length. Examples of string expressions include:

```
    "TML BASIC"
    str$
    "Apple" + gs$
    LEFT$(a$,5)
    MID$(UCASE$(a$),4,6)
```

Numeric expressions consist of numeric constants, numeric variables and numeric functions, optionally combined with the several numeric operators. Numeric expressions may evaluate to any of the five TML BASIC numeric types (integer, double integer, long integer, single-precision real, double-precision real) or to the special internal *SANE Extended-precision real* type. Examples of numeric expressions include:

```
123
123 + 4
myInt
COS(ang)
SQR((a^2) + (b^2))
```

Operators

Operators are special characters or reserved words that represent some arithmetic, relational, logical or string operation to perform. TML BASIC provides an extensive collection of operators that allow programs to perform just about any operation.

Operators must have *compatible* operands or else TML BASIC reports a Type Mismatch error. That is, a numeric operator requires both of its operands be a numeric value, while a string operator requires its operands be string values.

When an operator is used that has operands of different numeric types, TML BASIC automatically converts the numeric value with less precision/range to a value of the larger precision/range. For example, consider the following expression which uses the addition operator to add the value of an *integer* variable to a *double integer* variable:

```
myInt% + myDblInt@
```

Before TML BASIC, performs the addition, it automatically converts the value of *myInt%* to a *double integer* and then performs the addition.

There are four classes of operators: *arithmetic, relational, logical* and *string*. Each of these classes of operators is discussed in the following sections.

Arithmetic Operators

The *arithmetic operators* provided in TML BASIC perform the traditional mathematical operations for numeric values. Table 7-5 lists each of the arithmetic operators along with their respective operation.

Table 7-5
TML BASIC Arithmetic Operators

+	Identity (unary operator)
-	Negation (unary operator)
^	Exponentiation
*	Multiplication
/	Floating-point division
DIV	Integer division
MOD	Modulo (Integer only)
REMDR	SANE remainder
+	Addition
-	Subtraction

Note that TML BASIC provides two different division operators – integer and floating-point. The DIV operator is provided for efficient integer division. The DIV operator always converts its operands to integers and then performs the division to produce an integral quotient with no remainder. To obtain the remainder of an integer division, the MOD operator can be used. Like the DIV operator, its operands are always converted to integers before the operation is performed.

In TML BASIC, the evaluation of an arithmetic operation may sometimes cause an *overflow error*. An overflow error occurs when the result of an operation produces a value which is outside the storage capacity of the numeric type being used. For example, the code fragment

```
x% = 20000
x% = x% + 25000
```

overflows the storage capacity of the *integer type* (-32,768 to 32,767) since the result of the addition operator in the second statement is 45,000. Of course, addition is not the only arithmetic operator which can produce an overflow error. Consider the code fragment:

```
x% = - 20000
x% = x% - 25000
```

In this example, the result of the subtraction is -45,000 which again is outside the capacity of the *integer type*. In the two examples above, the error can be avoided by using a numeric type with a larger storage capacity, such as *double integers*.

It is also possible to cause an error by performing division by zero. This can occur when the second operand of the /, DIV and MOD operators is zero, and when the exponentiation operator (^) is used to raise zero to a negative power.

If a program is compiled with the Debug preference turned on (see Chapter 6 and Appendix B), TML BASIC generates code so that both of these types of errors are detected. The action taken when an overflow or division by zero error occurs depends upon whether the program contains an ON ERR statement which has been executed. If a program contains such a statement, control will transfer to the statement list after the ON ERR statement to process the error. Otherwise, execution of the program aborts and the appropriate runtime error message is reported.

If a program is compiled with the Debug preference turned off, an overflow or division by zero error will go undetected.

Relational Operators

The *relational operators* in TML BASIC allow programs to compare two values. The result of a comparison is a *Boolean* value which is either true or false. The result of a comparison is typically used to make a decision regarding program flow using the IF or DO...WHILE...UNTIL statements.

Since TML BASIC does not have a special boolean type, the values true and false are expressed as integer values. Any non-zero value is considered true, while the value zero is considered false. TML BASIC uses the non-zero value one (1) to represent the value of true for the relational operators. Thus, the expression 3=3 is true and has the value of 1, while 3=4 is false and has the value of 0. Strings are also considered to have a boolean value. If the string contains one or more characters then it is considered to have the boolean value true, and null strings (strings with zero characters) are considered to have the value false.

Table 7-6
TML BASIC Relational Operators

=	Equality
<> or ><	Inequality
<	Less than
>	Greater than
<= or =<	Less than or equal to
>= or =>	Greater than or equal to
<=>	Ordered (vs. unordered)

Note that when arithmetic and relational operators appear in the same expression, the arithmetic operators are evaluated first. For example, the following expression evaluates to true if A minus B is less than C plus D.

```
A - B < C + D
```

Logical Operators

The *logical operators* perform logical (Boolean) operations. Table 7-7 lists the logical operators available in TML BASIC.

Table 7-7
TML BASIC Logical Operators

NOT	Logical complement
AND	Conjunction
OR	Disjunction (inclusive or)
XOR	Exclusive or

The following table illustrates the behavior of the logical operators. The variables x and y may be any compatible type. See the section on relational operators for a discussion of what values constitute *true* and *false.*

Table 7-8
Values Returned by the Logical Operators

x	y	NOT x	x AND y	x OR y	x XOR y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

String Operators

There is only one operator in TML BASIC which returns a string value. This is the concatenation operator which is represented by the plus symbol (+). This is the same symbol used for addition when its operands have numeric values. Concatenation is the process of combining two strings together to make one string. For example, the following code fragment shows how to combine a volume name with a filename to create a file's complete *pathname*.

```
volName$  = "/TML/"
fileName$ = "STRINGS.BAS"
pathName$ = volName$ + fileName$
```

Remember that TML BASIC strings are limited to 255 characters. If a program attempts to create a string which is longer than 255 characters; TML BASIC will not perform the concatenation, but generate the String Too Long error.

Strings may also be used with the relational operators. String comparisons are performed by taking the corresponding characters from each string operand and comparing their ASCII codes. If the ASCII codes are the same for all the characters in both strings, the strings are considered equal. If the ASCII codes differ, the string containing the lower ASCII code is considered less than the other. If the end of one string is reached before the other then the shorter string is considered less than the other if they have been equal up to that point. The ASCII codes are listed in Appendix E. The following relational operations are ALL true:

```
"A" = "A"
"A" < "a"
"aa" > "aB"
"a" <= "aaaa"
```

If a program must compare two strings without regard for the case of the alphabetic letters, then the UCASE$ function should be used (see Chapter 10).

Precedence

In evaluating expressions which contain more than one operator, TML BASIC uses a set of precedence rules in order to determine which operator to evaluate first, and thus, which operands belong to which operators. TML BASIC defines three rules of precedence.

1. In an expression with more than one operator, the operator with the highest priority is evaluated first.

2. If an expression contains two or more operators of the same priority, then they are evaluated in order, from left to right.

3. The use of parentheses always overrides the priority of an operator to force a specific order of evaluation.

Table 7-9 lists each of the TML BASIC operators from highest to lowest priority.

Table 7-9
TML BASIC Operator Priority

+, -, NOT
^
*, /
DIV
MOD, REMDR
+, -
=, <>, ><, <, >, <=, =<, >=, =>, <=>
AND
OR, XOR

Chapter 8
Subroutines, Procedures, Functions and Libraries

Subroutines, procedures, functions and libraries provide the mechanisms in TML BASIC to divide and organize a program's source code into a more organized and more maintainable structure. They are often used to organize a collection of statements that must be frequently executed throughout a program.

A *subroutine* is a labeled set of statements executed when a GOSUB statement specifying the label is executed. A procedure is a named collection of code that behaves much like a subroutine, except that it provides for additional advanced programming features. A function is also a named collection of code like a procedure; but when executed returns a value. The value can then be used in an expression.

Procedures and functions in TML BASIC provide significant advantages over the traditional technique of organizing programs using GOSUB/RETURN. Procedures and functions offer the capability of parameters, local variables and recursion. If you have never programmed with these type of language features, you should make an effort to do so and discover their programming power.

Finally, a *library* is a special source code construct that groups together procedure and function declarations so that they can be compiled separate from any program. A library can be thought of as a *repository* for code which is used by one or more programs.

The next several sections discuss the implementation of subroutines, procedures, functions and libraries in TML BASIC and programming issues you should be aware of when using these language features.

Subroutines

A subroutine represents the traditional technique for BASIC programmers to organize a program into computational chunks. A subroutine begins with a label followed by a group of statements ending with the RETURN statement. A subroutine is executed by using the GOSUB statement.

A GOSUB statement indicates that execution should temporarily suspend at the current statement and control should transfer to the statement indicated by the label in the GOSUB statement. When the RETURN statement is encountered, the subroutine terminates and control returns to the statement immediately after the calling GOSUB. The following code fragment illustrates the use of a subroutine.

```
GOSUB CalculateGrade
PRINT Grade
END

CalculateGrade:
   Total = 0
   FOR i = 1 to NumGrades
      Total = Total + Grades(i)
   NEXT i
   Grade = Total / NumGrades
RETURN
```

A subroutine may call other subroutines, which in turn may call yet other subroutines. TML BASIC keeps track of where execution should resume when the RETURN statement is encountered with a data structure called the *Runtime Stack*. Each time a GOSUB statement is executed, TML BASIC pushes the *program counter* for the statement immediately after the GOSUB onto the stack. Then, when a RETURN statement is encountered, the program counter is removed from the stack and made the *current program counter* – the place where execution continues.

In some special cases, a program may not want to return from a subroutine back to the caller. This might happen when an error occurs, and a program decides it should continue execution elsewhere. To do this, the program must first remove the program counter value stored in the Runtime Stack by using the POP statement. When the POP statement is executed it removes the program counter from the runtime stack, and then continues execution with the next statement after the POP.

When using the RETURN and POP statements, your program must be certain that a corresponding GOSUB has been executed so that a program counter value has been placed on the Runtime Stack. If your program attempts to execute a RETURN or a POP without a matching GOSUB, the runtime error "RETURN/POP without matching GOSUB" occurs. If you compile a program with the Debug option turned off then this error will go undetected, and the execution of your program will certainly go astray.

A subroutine may also be called using the ON...GOSUB statement. The ON...GOSUB statement works just like the normal GOSUB statement except that it chooses among several subroutines to call depending upon the value of an expression. For example, the following statement calls the *IssuePayCheck*, *IssueBonusCheck* or *IssueExpenseCheck* subroutine depending upon the value of the variable *doCheck%*.

```
ON doCheck% GOSUB IssuePayCheck,IssueBonusCheck,IssueExpenseCheck
```

If the value of *doCheck%* is 1 then the first named subroutine is called – *IssuePayCheck*, and if its value is 2 the second named subroutine is called, etc. If the value of the expression is 0 or greater than the number of named subroutines, the statement is skipped. As with the GOSUB statement, control returns to the statement immediately after the ON...GOSUB when a RETURN is executed.

For more information about the TML BASIC Runtime Stack see the section "A Lesson on Stacks" later in this chapter, and the discussion of the Preferences dialog in Chapter 6. Also read about the $StackSize metastatement in Appendix B.

Procedures

Procedures are a group of statements that are surrounded by the DEF PROC and END PROC statements. A procedure behaves much like a subroutine except the PROC statement is used to call a procedure. When a procedure is called using the PROC statement, execution temporarily suspends at the current statement and control is transferred to the procedure. When the procedure returns, execution continues with the first statement after the PROC statement.

Procedures however, provide for several additional language features: zero or more parameters, local variables and local labels. These features allow you to write blocks of code which are isolated from the rest of the program. The ability to hide the names of parameters and local variables from the main program allows you to use names that are also used in the main program, but which retain separate values and do not affect those in the main program. Local label names can also be the same as labels in the main program (or other procedures) since they are also hidden.

The use of local variables and labels allows for true modular design of your programs. In fact, you can create procedures that work in several different programs without considering the issue of duplicate variable names. TML BASIC provides a powerful language feature for sharing procedures (and functions) between several programs – the Library. Libraries allow you to group useful procedures and functions together and then automatically provide them to several different programs. Libraries are discussed later in this chapter.

Defining Procedures

A *procedure definition* begins with the DEF PROC statement and continues until a matching END PROC statement. The DEF PROC statement is required to be the first statement on a line. Using the notation defined in Chapter 7, the general syntax for defining a procedure is as follows:

```
DEF PROC procedurename [ ( parameter { , parameter } ) ]
   LOCAL variable { , variable }
   •
   • statements
   •
END PROC [ procedurename ]
```

The *procedurename* declares the name of the procedure. The name for the procedure must follow the same rules for names as variables, and must not appear in any other DEF PROC statements. Procedure names do not have a type character at the end of their names. The following are example procedure names:

```
DEF PROC MyProc
DEF PROC Proc123
DEF PROC Do.It
```

Following *procedurename* is the optional formal parameter list. The formal parameter list defines the names and types of variables which are passed to the procedure when it is called. Parameters are separated by commas and they can be any type of simple variable. Parameters receive their values when the procedure is called, thus, they do not retain their values between calls to the procedure. If a procedure must retain the values of variables between calls, it must use global variables or arrays.

Each parameter becomes a local variable when a procedure is called and has an initial value equal to the corresponding actual parameter given in the PROC statement. Additional local variables may be defined using the LOCAL statement described below. Procedures are limited to 16 parameters.

The statements between the DEF PROC and END PROC are called the body of the procedure. When the procedure is called, execution begins with the first statement after the DEF PROC and continues until the matching END PROC. Within the body of the procedure it is illegal to define another procedure or function. It is also not possible for a GOTO or a GOSUB statement to branch to a label outside of the body of the procedure; however, other procedures and functions may be called. TML BASIC also restricts the use of the DIM statement within the body of a procedure to dynamic arrays (see Chapter 7).

Local Variables

Procedures (and functions) provide a mechanism for defining temporary variables which are created when the procedure is called and destroyed when the procedure completes and returns to the caller. These variables are called *local* variables. Variables which are not local variables are called *global* variables because they exist for all procedures, functions and the main program to use.

As mentioned above, parameters are equivalent to local variables except they have initial values which correspond to the matching actual parameters. Additional local variables are created with the LOCAL statement. Following is an example of the LOCAL statement.

```
DEF PROC myProc
  LOCAL anInt%, aDblInt@, aString$
  LOCAL anotherString$
   •
   • statements
   •
END PROC
```

A procedure may contain zero or more LOCAL statements and each LOCAL statement may declare one or more variables. However, TML BASIC restricts the use of the LOCAL statement. The LOCAL statements in a procedure must appear after the DEF PROC, but before any other statement. The only exception is that the REM statement may appear before a LOCAL statement.

When a procedure is called, storage for the local variables is created on the *Runtime Stack*. Local numeric variables are initialized with the value zero and local string variables are initialized with the null string. When the procedure returns control to the calling statement, storage for the local variables is deallocated. Thus, local variables do not retain their values between procedure calls. For example the following procedure will always print an empty line regardless of how many times it is called.

```
DEF PROC SillyProc(newMsg$)
   LOCAL Msg$
   PRINT Msg$
   Msg$ = Msg$ + newMsg$
END PROC
```

When a value is assigned to a variable which is not declared as a local variable, it assigns the value into the global variable with the given name. If the global variable does not exist, TML BASIC automatically creates the global variable and then assigns the value. It is possible to guarantee that an assignment statement references a local variable using the FN form of the assignment statement.

```
FN anyVar = expression
```

By preceeding the assignment statement with the reserved word FN, TML BASIC checks to guarantee that the variable *anyVar* is declared as a local variable, otherwise it gives the error "Variable is not LOCAL".

Using Procedures

A procedure is called by using the PROC statement in a program. The PROC statement specifies the name of the procedure to call followed by any required parameters in parentheses. For example:

```
PROC PrintLine("Totals",TotalDebits@,TotalCredits@)
```

In this example, the procedure *PrintLine* has three parameters – one string parameter, followed by two double integer parameters. To call the procedure, TML BASIC first creates the procedure's formal parameters and then assigns them with the values of their matching actual parameters. Control is then passed to the procedure *PrintLine*. When the procedure returns, execution continues with the first statement after the PROC statement.

Functions

There are two types of functions in TML BASIC: *single-expression (simple) functions* and *multiline functions*. Functions are used to group together one or more statements that compute and return a value. Single-expression functions contain only one expression for computing the value of the function. These are the traditional style functions found in older BASIC implementations such as AppleSoft BASIC. Multiline functions are constructed similar to procedures except that the DEF FN and END FN statements are used to group the function's statements. Both the single-expression and multiline functions can have formal parameters. Additionally, multiline functions can have local variables.

Defining Functions

As mentioned above, TML BASIC supports two types of functions: single-expression and multiline. A single-expression function is one whose code is contained on a single line of source code. The general sytnax for a single-expression function is as follows:

```
DEF FN functionname [%|@|&|#|$] [ ( parameter { , parameter } ) ] = expression
```

The *functionname* declares the name of the function. Following the function name is an optional type character used to specify the type of the function's result value. If no type character is given the function returns a single-precision real value. A function may optionally have a sequence of parameters whose values are used to compute the value of the function. TML BASIC limits a function to 16 parameters.

The *expression* specifies the computation to be performed when the function is called. If the function returns a string value, the *expression* must evaluate to a string. Likewise, if the function returns a numeric value, the *expression* must evaluate to one of the numeric types. The following are examples of simple single-expression functions which convert between degress Celcius and degrees Fahrenheit.

```
    DEF FN CtoF(degreesC) = 1.8 * degreesC + 32
    DEF FN FtoC(degreesF) = (degreesF - 32) * 0.555555
```

The second type of function supported in TML BASIC is the multiline function. The general syntax for defining a multiline function is as follows:

```
DEF FN functionname [%|@|&|#|$] [ ( parameter { , parameter } ) ]
   LOCAL variable { , variable }
   •
   • statements
   •
   FN functionname [%|@|&|#|$] = expression
   •
END FN [ functionname ]
```

The syntax for a multiline function is exactly the same as a procedure with two exceptions. First, the function name is optionally followed by a type character which specifies the type of the function's result value. Second, the statements in the body of the function must contain at least one assignment statement which gives the function its value.

The assignment statement which gives the function its value must begin with the reserved word FN followed by the function's name and optional type character. As with single-expression functions, the *expression* assigned to the function must be compatible with the function type. The following is an example of a multiline function which computes the factorial of a number.

```
DEF FN Factorial#(n%)
   LOCAL total#
   IF n% < 0 THEN
      FN Factorial# = 1
   ELSE
      FOR i% = n% TO 2 STEP -1
         total# = total# * i%
      NEXT i%
      FN Factorial# = total#
   END IF
END FN Factorial#
```

Using Functions

Functions return values. As such, functions are used within expressions to compute values. A function is called by using the FN reserved word followed by a function name. The function name is then followed by the appropriate number of actual parameters enclosed in parentheses. For example:

```
tempF = 100
tempC = FN FtoC(tempF)
PRINT tempF, tempC
```

In this example, the function *FtoC* is called to convert degrees Fahrenheit to Celcius. After the reserved word FN, the function name *FtoC* along with its parameter.

Formal versus Actual Parameters

Procedures and functions actually have two types of parameters: *formal* and *actual*. The parameters that appear in a parameter list of a DEF PROC or DEF FN statement are called formal parameters. Formal parameters are essentially local variables of the defined procedure or function and are completely separate from the rest of a program. To illustrate this, consider the following statements:

```
height% = 16
DEF FN Perimeter%(height%,width%) = 2 * height% + 2 * width%
PRINT height%, FN Perimeter%(8,4), height%
```

The variable *height%* in the first and third lines is unrelated to the formal parameter *height%* in the function definition in the second line. When these statements are executed, the value of *height%* in the third line is unaffected by the function call.

The parameters of the function *Perimeter%* in the third line are actual parameters. The values of the actual parameters are implicitly copied into the formal parameters when the function is called. If the value of a formal parameter changes during execution of the function, the value of the actual parameter remains unchanged.

Program Flow

The location of a procedure or function definition in the source of a program is not important. The PROC or FN statements can be used to call a procedure or function anywhere in the program even if the procedure or function is declared later in the source code. Before TML BASIC compiles a program, it first scans the source file for all DEF PROC and DEF FN statements and records their names and the number and type of parameters. Thus, when a procedure or function is called using the PROC or FN statements, TML BASIC knows if the procedure or function has been defined and if the correct number of parameters have been given.

A program need not direct its flow of control around procedure and function definitions. When executing statements in the main program and a DEF PROC or DEF FN statement is encountered, the next statement executed is the first statement which appears *after* the corresponding END PROC or END FN. The DEF PROC...END PROC and DEF FN...END FN act as large comments around its enclosing statements. The only way to execute the statements of a procedure or function is to call it using the PROC or FN statements respectively.

Consider the following code fragment:

```
PROC PrintMessage
DEF PROC PrintMessage
   PRINT "Hello"
END PROC
PRINT "Goodbye"
END
```

The output of this code is

```
Hello
Goodbye
```

and not

```
Hello
Hello
Goodbye
```

The second "Hello" is not output because control passes completely around the *PrintMessage* procedure and continues with the PRINT "Goodbye" statement.

Recursion

TML BASIC procedures and functions are recursive. By *recursive* we mean that a procedure or function may call itself, or it may call another procedure which in turn calls the same procedure or function.

The following is an example of how to write a recursive multiline function which computes a factorial.

```
DEF FN Factorial(n%)
   IF n% > 1 THEN
      FN Factorial = n% * Factorial(n%-1)
   ELSE
      FN Factorial = 1
   END IF
END FN Factorial
```

As you can see, recursion is a powerful mechanism for expression algorithms. In the case of the Factorial function, the factorial of a number is computed without any loops or local variables.

Writing programs which use recursive procedures and functions can sometimes require that you increase the size of the runtime stack. Since every time a procedure or function is called, its parameters, local variables and return address is placed on the runtime stack, it is possible to exhaust the available runtime stack space in a program that uses recursion extensively. To increase the runtime stack size use the $StackSize metastatement or change the K-byte Stacksize option in the Preferences dialog. See the section "A Lesson on Stacks" below for more information.

While recursion is a very powerful programming technique, it can also be the source of complicated program errors. The most common error when using recursion is omitting a termination condition. By not providing a termination condition for the recursion, a procedure or function will continue to call itself over and over again, until finally memory has been exhausted and the machine crashes. In the Factorial function above, the recursion terminates when the parameter *n%* is less than or equal to 1.

A Lesson on Stacks

Before leaving this chapter, a discussion concerning the *Runtime Stack* used by TML BASIC programs is in order. As mentioned in the previous sections of this chapter, TML BASIC provides for user defined subroutines, procedures and functions. Procedures and functions may have parameters and local variables and may also be called recursively.

To support these language features, TML BASIC implements a data structure called the Runtime Stack. This stack is used to save the program counter of the statement immediately following the GOSUB, PROC and FN statements so that when these routines return, the program knows where to resume execution. The stack is also used to reserve storage for procedure and function parameters and local variables. If a procedure or function is called recursively, then multiple copies of the parameters and local variables exist in the Runtime Stack.

The runtime stack does not have an unlimited size. In fact, the default size allocated for the runtime stack is 8K bytes. While this amount of storage is quite sufficient for most all TML BASIC programs, it is possible that a program which is highly recursive and declares many parameters and local variables could exceed this size.

TML BASIC provides two ways to change the size of the runtime stack created for programs. In the Preferences dialog you can change the *edit-text item* which appears next to the message "K-byte Stack" as described in Chapter 6. After this change has been made, all subsequent compiles will create a runtime stack of the new size. The second technique, and the preferred technique for programs which need a larger stack, is to use the TML BASIC $StackSize metastatement (see Appendix B). If a program uses the $StackSize metastatement, it overrides the value specified in the Preferences dialog, ensuring the required stack size is allocated.

The amount of memory that can be requested for the runtime stack is not unlimited. The smallest size that can be requested for the runtime stack is 1K bytes, and the largest is 32K bytes. This number is very large for a runtime stack, and no program should ever need to request that size. The TML BASIC runtime stack is allocated in Bank 0 of the Apple IIGS memory. The total available memory in this bank is approximately 40K bytes, and since this memory is required for many other uses as well, it is not wise to wastefully allocate more memory than is needed by your program.

TML BASIC also provides a mechanism which allows you to determine if a program is using more space for the runtime stack than has been allocated to it. If you select the Check Stack Size option in the Preferences dialog then TML BASIC will generate code which checks the available runtime stack space everytime a procedure or function is called. In there is not sufficient space left in the runtime stack to successfully make the call, the runtime error "Stack Overflow" occurs. To determine the stack size required for a particular program, repeatedly increase the "K-byte Stack Size" option in the Preferences dialog until the program executes without causing the "Stack Overflow" error.

Libraries

A library is a special source code construct that groups together procedure and function declarations so that they can be compiled separate from any program. TML BASIC allows code which is compiled in a library to be used in other libraries and in programs just as if the code were textually included in the source code file. A library can be thought of as a *repository* for code.

Libraries provide two major benefits. First, libraries allow for easy code *sharing* among different programs. A collection of commonly used procedures and functions can be grouped together in a library and then provided to one or more different programs. Second, libraries provide a more manageable way to create large programs. Using libraries, it is possible to partition your program into smaller pieces which can be developed and tested separately from the rest of the program. Because libraries are compiled separately, they also keep you from having to recompile an entire program when only changing a small part of the source code.

Libraries are also used by TML BASIC to define the interfaces to the Apple IIGS Toolbox. Libraries and the Toolbox are described in detail in Chapter 11.

The source code for a library is almost exactly the same as a program, except for two simple rules. First, the source code must begin with the DEF LIBRARY statement and end with the END LIBRARY statement. The DEF LIBRARY statement must be the first statement in the source code since it is the only means that TML BASIC uses to distinguish between a program and a library. And second, the only legal statements that may appear in the body of the library are REM, DIM, DEF PROC and DEF FN. Any legal BASIC statement may appear within the body of a procedure or multiline function declaration.

The following is an example of a simple library that implements a fixed size integer stack.

```
DEF LIBRARY IntegerStack

   REM This library implements a simple 100 element integer stack.

   DIM theStack%(99)                 'Declare the stack

   DEF PROC ClearStack               'Make the stack empty
      stackTop% = -1
   END PROC ClearStack

   DEF PROC Push(aValue%)            'Push a new value onto the stack
      IF stackTop% < 99 THEN
         stackTop% = stackTop% + 1
         theStack%(stackTop%) = aValue%
      END IF
   END PROC Push

   DEF FN Pop%                       'Remove the top element of the stack
      IF stackTop% >= 0 THEN
         FN Pop% = theStack%(stackTop%)
         stackTop% = stackTop% - 1
      END IF
   END FN Pop%

END LIBRARY
```

This simple library implements two procedures, a function and declares two global variables. Note that only the REM, DIM, DEF PROC and DEF FN statements are used in the body of the library, but that assignment, IF statements or any other statements are allowed in the body of a procedure or function in the library. No other statements are allowed in the body of the library because a library can not be *run* like a program. The only code that is executed in a library are procedures and functions which are called by a program.

Using a Library

The code defined in a library can be used by other libraries or a program. The LIBRARY statement is used to make the code from a library available in a program. When the LIBRARY statement appears in a program, TML BASIC behaves as if all of the code in the specified library were actually in the program at the point where the LIBRARY statement occurs. Thus, a program can call the procedures and functions defined in the library or use any of its global variables.

For example, the following program uses the *IntegerStack* library defined above.

```
LIBRARY "IntegerStack"

PROC ClearStack

DO
   HOME
   PRINT " Integer Stack Demo"
   PRINT "    (1) Push an integer"
   PRINT "    (2) Pop an integer"
   PRINT "    (3) Quit"
   INPUT "Enter an option: "; option%

   IF option% = 1 THEN
      INPUT "Enter the integer: "; newValue%
      PROC Push(newValue%)
   ELSEIF option% = 2 THEN
      PRINT "The top stack element was: "; FN Pop%
   END IF
UNTIL option% = 3
END
```

By simply including the LIBRARY statement at the beginning of the program, all of the declarations in the *IntegerStack* library are available to the program. In addition, the same library can be used by different programs without having to edit or recompile the *IntegerStack* library.

You should use libraries to develop your own collections of useful procedures and functions which can be shared among the types of programs you write.

Compiling Libraries

Because a library is not a program, it can not be *run*. Thus, the behavior of the TML BASIC **To Memory & Run** and **To Disk** compile commands is different than compiling programs. When you choose to compile a library to memory using the **To Memory & Run** command, TML BASIC compiles its source code, and then returns to the editor without executing the source code. The compiled source code is retained in memory so that other programs can use the code without having to recompile the library.

The **To Disk** command compiles the library's source code and then creates a .LIB file which contains a permanent copy the compiled code for the library. Thus, when a program specifies a particular library in the LIBRARY statement, and its compiled code is not in memory, TML BASIC can read it directly from the disk without having to recompile the library's source code over again.

For more information about compiling techniques, see Chapter 3.

Predefined Libraries

TML BASIC provides a collection of predefined libraries which define the interface to the Apple IIGS Toolbox. Using these libraries, BASIC programs may call the Toolbox to use graphics and sound, create menus and windows, etc. These libraries are described in Appendix C and found in the LIBRARIES folder of the TML BASIC distribution disk.

See Chapters 11 through 13 for information on how to use these libraries for programming the Toolbox.

Chapter 9
Files

This chapter discusses the language features and capabilities of TML BASIC for manipulating disk files and devices. If you are unfamiliar with the concepts of files, volumes, disks and ProDOS you should study the *Apple IIGS Owner's Guide* for an introduction to the ProDOS operating system.

ProDOS 16 Fundamentals

ProDOS 16 is the operating system for the Apple IIGS. As such, it is responsible for implementing much of the interface between Apple IIGS hardware and applications. It manages the creation and modification of files. It accesses the disk devices on which files are stored and retrieved. It dispatches interrupt signals to interrupt handlers. It also controls certain aspects of the Apple IIGS operating environment, such as pathname prefixes and routines for quitting programs and starting new ones.

Filenames

A *disk file* is an ordered collection of information is stored on a disk and has several attributes, including a *filename* and a *filetype*. Because TML BASIC operates within the ProDOS 16 operating system, the file naming conventions and operations in TML BASIC must abide by the rules of ProDOS 16.

A *filename* for a disk file in TML BASIC can be any sequence of 15 or fewer letters (A-Z and a-z), digits (0-9) or periods (.); where the first character in a filename must be a letter. If a program attempts to use a filename longer than 15 characters, or a name that contains an illegal character, an error will occur.

Peripheral devices such as the keyboard, screen and printer connected to the Apple IIGS are also treated as files in TML BASIC. These are called *character files* rather than *disk files.* The naming conventions for character files is the same as disk files, except that the name begins with a period (.). For example, ".PRINTER" refers to the the printer device connected to slot 1. TML BASIC predefines some character filenames, and a program can define others using the ASSIGN statement. See Chapter 10 for a discussion of the ASSIGN statement and the predefined character filenames. By treating peripheral devices as files, TML BASIC provides a single consistent method for performing input and output in a program.

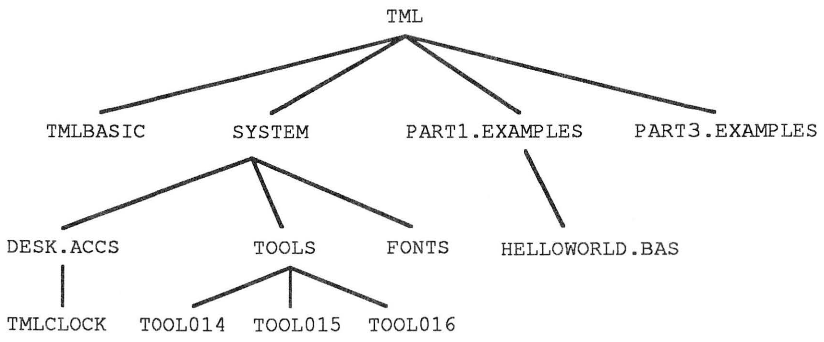
The filetype attribute is a special integer value which indicates the contents of a file. There are filetypes which indicate text files, ProDOS 16 directories, BASIC Data Files, etc. Appendix F provides a list of the most common filetypes.

To open a file for input and/or output, the OPEN statement is used. The OPEN statement associates a character or disk file with a *filenumber* which is used by TML BASIC file input and output statements. TML BASIC supports up to 32 open files in a program, numbered from 0 to 31. File number 31 has a special purpose in TML BASIC. It is used by the CATALOG statement to read a disk directory. If a program has all 32 files opened and executes the CATALOG statement, an error will occur. Currently, ProDOS 16 version 1.x only allows up to eight files to be opened at once, however, when new versions of ProDOS 16 become available which support more than 8 open files, TML BASIC will be able to open up to 32 files.

Pathnames

The ProDOS 16 filetypes include the special *directory* filetype. A directory file contains the names and disk locations of other files on a disk. A directory file can contain other directory files, thus creating a hierarchy for the file organization on a disk. These nested directories are sometimes called *subdirectories*. Every volume contains at least one directory called the *root directory*. The root directory file has the same name as the volume. All other files and directories are contained in this directory.

The following figure illustrates part of the directory organization of the TML BASIC distribution disk.



To access a file using ProDOS 16, more than just its filename is required. ProDOS 16 requires a *pathname* to fully specify a file. A pathname is merely a series of filenames, each preceded by the slash (/) character. The first filename in a pathname is the root directory's filename (volume name). Successive filenames indicate the path, from the root directory to the file that ProDOS 16 must follow to find a particular file. For example, the pathname for the the file TMLCLOCK in the above diagram is as follows:

```
/TML/SYSTEM/DESK.ACCS/TMLCLOCK
```

The maximum length for a pathname is 64 characters.

A pathname which begins with the filename of the root directory (volume name) is called a *full pathname*. Files can also be designated with a *partial pathname*. A partial pathname is a portion of a pathname that does not begin with a root directory name and does not begin with the slash character. The following are partial pathnames for the file TMLCLOCK.

```
TMLCLOCK
DESK.ACCS/TMLCLOCK
SYSTEM/DESK.ACCS/TMLCLOCK
```

Whenever a partial pathname is used, ProDOS 16 automatically adds a *prefix* to the partial pathname to create a full pathname. A prefix is a pathname that indicates a directory. A prefix always begins with a slash and a root directory name followed by zero or more directory names. The following are legal prefix pathnames from the example above:

```
/TML/
/TML/SYSTEM/
/TML/SYSTEM/DESK.ACCS/
/TML/PART1.EXAMPLES/
```

The slashes at the end of these prefixes are optional, but are helpful reminders that these are prefix pathnames and not full pathnames to the respective directory files.

ProDOS 16 stores eight prefixes numbered 0 through 7. Prefix number 0 is called the *default prefix*. Whenever a partial pathname is given, ProDOS 16 automatically joins prefix 0 to the front of the partial pathname to create a full pathname. A prefix pathname can be a maximum of 64 characters long. Since partial pathnames can also be up to 64 characters long, it is possible to create pathnames up to 128 characters long. The TML BASIC modifiable reserved variable PREFIX$ contains the value of the ProDOS 16 prefix 0. TML BASIC also provides the PREFIX statement and the PFX$ function for manipulating prefixes.

Finally, it is possible to override the use of prefix 0 when using partial pathnames and designate any prefix by preceding a partial pathname with a prefix number and a slash character. For example:

```
1/DESK.ACCS/TMLCLOCK
6/HELLOWORLD.BAS
```

Manipulating Files

TML BASIC provides several statements which provide direct access to the ProDOS 16 operating system for manipulating files. These statements allow programs to create and delete files, rename files, lock and unlock files, catalog a directory, and determine what volumes are available. Each of these statements is discussed in detail in Chapter 10, however, a quick review of these operations is provided below.

CREATE Statement

The CREATE statement is used to create disk files. The CREATE statement can be used to create directories, text files, BASIC data files, and any other valid ProDOS 16 filetype. The following is the syntax for the CREATE statement:

```
CREATE Pathname [,FILTYP= DIR|TXT|SRC|BDF|Filetype [,SubType]]
```

The reserved word CREATE is followed by the pathname of the file to be created. Optionally, a pathname may be followed by the filetype specification and subtype specification. If the optional filetype specification does not appear in a CREATE statement, a text file is created. The following table shows the predefined filetype names, their alternate names and meaning.

Table 9-1
FILTYP= names

Filetype Mnemonic	Alternate Mnemonic	Meaning
DIR	CAT	Subdirectory
TXT	TEXT	Text file
SRC		Source file
BDF	DATA	BASIC Data File

Appendix F contains a list of the most often used ProDOS 16 filetypes.

If the FILTYP= argument appears in a CREATE statement, it may optionally be followed by a file subtype specification. The subtype is an unsigned integer value in the range 0 to 65,535. If the subtype is not specified, the default value of zero is used except for the case of BASIC Data Files. If the specified file type is a subdirectory (DIR) then the subtype is zero regardless of the value specified. The meaning of the subtype varies depending upon the file type.

BASIC Data Files require the subtype value be in the range 3 through 32,767. The reason for this is that TML BASIC uses the subtype of a BASIC Data File as the file's logical record size. The logical record size of a BASIC Data File must be known in order to support random-access to the file's records. See the "Accessing BASIC Data Files" section later in this chapter for more information about BASIC Data Files.

An attempt to create an already existing file using the CREATE statement causes the "Duplicate File Error" to occur.

The following are three examples of the CREATE statement. The statements create a directory, a text file and a BASIC Data file respectively.

```
CREATE "/TML/MY.EXAMPLES", FILTYP=DIR 'Create a new subdirectory

CREATE "GRADES"                       'Create a text file

CREATE "MYROLODEX",FILTYP=BDF,100     'Create BASIC Data File
                                      ' with record size (subtype) of 100
```

DELETE Statement

The DELETE statement is used to remove a disk file from a volume. A directory file can only be deleted if all the files in the directory have been deleted. It is, of course, impossible to delete the *root* directory. The following is the syntax of the DELETE statement:

```
DELETE Pathname
```

Any number of errors may occur when using the DELETE statement if the file is currently in use, locked, on a write protected disk, etc. See Appendix A for a complete list of the possible runtime errors.

RENAME Statement

The RENAME statement is used to change the name of a volume, directory or any other file. The syntax for the RENAME statement includes the old pathname followed by a comma and then the new pathname.

```
RENAME OldPathname, NewPathname [,FILTYP= TXT|SRC|BDF|Filetype]
```

The *OldPathname* must be the name of an existing file, and the *NewPathname* may be any legal ProDOS 16 path. Using RENAME it is possible to change the name of a file and even move the file into a different directory; however, it is impossible to move a file to another disk by merely changing its pathname. For example:

```
RENAME "HELLOWORLD.BAS", "HELLO.BAS"    'Change file name
RENAME "HELLO.BAS", "/TML/HELLO.BAS"    'Change file's directory
```

If the optional FILTYP= argument is used, the filetype of the renamed file is changed as well. It is possible to change only the filetype of a file using the FILTYP= argument when the *OldPathname* and *NewPathname* are the same.

LOCK and UNLOCK Statements

The LOCK and UNLOCK statements are used to change a file's protection. The syntax for these statements includes only the reserved word LOCK or UNLOCK followed by the pathname of the file whose protection is to be changed.

```
LOCK Pathname
UNLOCK Pathname
```

The LOCK statement prohibits writing to, renaming or deleting the named file. Any filetype, including directories can be locked except for the root directory. The UNLOCK statement removes the protection placed upon a file by the LOCK statement.

CATALOG Statement

The CATALOG statement is used to display a listing of the files contained in a directory. The CAT statement is a short form of the CATALOG statement that only displays a subset of the directory information. Optionally, following the reserved word CATALOG can be the pathname of a directory.

```
CATALOG [ Pathname ]
CAT [ Pathname ]
```

If the pathname is a volume name, all the files in the volume's root directory are displayed. Otherwise, the pathname should specify the name of a subdirectory file, in which case all of its files are displayed. If the pathname is omitted, the pathname in the ProDOS 16 prefix 0 is displayed.

The CATALOG statement displays the filename, filetype, size, modification date, creation date and subtype for each file in a specified directory.

VOLUMES Statement

The VOLUMES statement is used to read the volume name for each ProDOS 16 device and display its name. The ProDOS 16 devices are numbered .D1 through .D9 inclusive. The display lists the device name, its volume name and the number of free bytes of storage available on the volume.

Opening and Closing Files

Before a program can read from or write to a file that has been created, it must be opened. After a program is finished accessing a file it should be closed. As noted before, TML BASIC allows up to 32 files to be open simultaneously, however, the current versions of ProDOS 16 (version 1.x) only support eight open files. Only later versions of the operating system will allow programs to take advantage of TML BASIC's ability to open up to 32 files.

OPEN Statement

The OPEN statement is used to open files for access, and must precede any file I/O routines accessing a given file. The following is the general syntax for the OPEN statement:

```
OPEN Pathname, [ FILTYP= DIR|TXT|SRC|BDF|Filetype ]
    [ FOR INPUT|OUTPUT|APPEND|UPDATE ] AS # Filenumber [, Recordsize ]
```

The minimum required arguments following the reserved word OPEN are the file's pathname followed by a comma, the reserved word AS and a file reference number. The file must have been previously created and must exist on a disk currently mounted in a disk drive. If a partial pathname is used, it is joined with prefix 0 to create the full pathname. The file reference number is used in all subsequent TML BASIC I/O statements for accessing the file. The following are some examples of the OPEN statement:

```
OPEN "HELLOWORLD.BAS", AS #10
OPEN "/TML/MYSTUFF/INVOICES", AS #20
OPEN ".PRINTER", AS #1
OPEN ".MODEM", AS #2
```

It is generally good programming practice to adopt a convention for the use of file reference numbers. One good convention is to use the file reference numbers 1 through 7 for character device files where the file reference number corresponds to its slot, and the numbers 10 through 31 for disk files.

The optional FOR clause in the OPEN statement is used to qualify the *access mode* for the file. The supported access modes are INPUT, OUTPUT, APPEND and UPDATE. If the FOR clause is not used, the file is opened for UPDATE. The FOR INPUT clause specifies that the file is opened for read-only access, and cannot be written to. For example:

```
OPEN myFile$, FOR INPUT AS #10
```

The FOR OUTPUT clause specifies that the file is opened for write-only access, and cannot be read from. For example:

```
OPEN myFile$, FOR OUTPUT AS #10
```

The FOR APPEND option is a variant of the FOR OUTPUT clause. It is used for sequential access (discussed later) and allows the PRINT# and WRITE# statements to append new information to the end of a file without disturbing any existing data in the file. For example:

```
OPEN myFile$, FOR APPEND AS #10
```

Finally, the FOR UPDATE clause is used to open a file for read-write access as long as the filetype supports such access. For example, you cannot read from a printer.

The optional FILTYP= clause of an OPEN statement is used to specify the type of a file. The FILTYP= clause is primarily used to ensure that a file being opened is of the expected filetype. If a program attempts to open a file using the FILTYP= clause and the file's type does not match the specified filetype, the file is not opened and an error is reported. Any of the predefined filetype names (see CREATE) can be used with the FILTYP= clause or an unsigned integer value.

The FILTYP= clause is also used with the OPEN statement to open files which have not been created. If the OPEN statement finds that the specified file does not exist, and the FILTYP= clause is given, it will implicitly call the CREATE statement first and then open the newly created file.

Finally, the optional *RecordSize* argument is used to specify the record size for random access to the file using the INPUT# and GET# statements for non-Basic Data Files. If the file being opened is an existing BASIC Data File, the record size argument is ignored and the record size used is the size specified when the file was created. For more information about random file access see the "Accessing Text Files", "Accessing BASIC Data Files" and "Accessing Binary Files" sections below.

CLOSE Statement

The CLOSE statement is used to close a file previously opened with the OPEN statement. After a file has been closed, no further access is possible. A program should always close a file after it has finished accessing it. The following is the syntax of the CLOSE statement:

```
CLOSE
CLOSE# FileNumber
```

The CLOSE statement alone closes *all* files which are currently open. In addition, TML BASIC closes all open files when the RUN and END statements are executed and when a program terminates.

A variation of the CLOSE statement, the CLOSE# statement, can be used to close a single file. With this statement a program specifies the file reference number of the file to be closed. For example:

```
    CLOSE #10
```

File Access Techniques

Each of the six file access statements discussed in the next three sections can be used for both *sequential* and *random* file access. Sequential file access is like reading a book; access begins at the beginning of the file and continues in order to the end of the file. Random file access on the other hand allows a program to read or write to arbitrary locations in the file.

The following paragraphs define the concepts of sequential and random file access as they relate to TML BASIC. For specific information on the different TML BASIC statements which access files, see the sections "Accessing Text Files", "Access BASIC Data Files", and "Accessing Binary Files" later in this chapter.

Sequential Access

TML BASIC stores a *current file position* for every file opened with the OPEN statement. When a file is first opened, the current file position is set to the beginning of the file, unless the file was opened with the FOR APPEND option, in which case the current file position is set to the end of the file. The current file position is the location where each TML BASIC I/O statement reads from or writes to a file.

Sequential access is the most common technique for file access. When a file is read from or written to, the file is accessed at the *current file position*. After the file is accessed, the current file position is updated to point to the very next data element in the file so that the next file access begins where the previous access left off. If a program is writing to the file and the current file position reaches the end of the file, the size of the file is extended by the size of the data being written to the file. After the data is written, the current file position is updated to point to the new end of file.

Sequentially accessed files, opened using the FOR UPDATE (the default) access mode, pose some interesting questions. Using this access mode the current file position is initially set to the beginning of the file. After the program writes new information to the file, where is the file's end of file, and what data does it actually contain? Is the end of file at the end of the original contents of the file or the new contents? Depending on the situation, the answer could be either the original or the new. If the original contents of the file have been fully overwritten, all of the original information is lost and the end of file is at the end of the new contents. If only a portion of the original information is overwritten then some of the original information still exists in the file and the end of file is at the end of the original contents of the file.

To avoid the problem of old file contents remaining in a file after it has been written to using the FOR UPDATE access mode, a program should first delete the existing file, and then re-create and open the file.

Random Access

In contrast to sequential access, random file access allows each of the TML BASIC I/O statements to specify a file *record number* as the new location of the file's *current file position* before file access occurs. In order to specify a record number, however, a file must first be organized into a collection of *records*. A record is a data structure consisting of a *fixed* number of bytes. The first record in a file is numbered zero, the second record is numbered one, etc. Each successive record lies adjacent to the next, with no intervening storage. Thus, a file containing N records, each B bytes large, contains records numbered in the range 0 through N-1, and a file size of N*B bytes.

When a file is written to using random access to a record which does not yet exist in the file, the file is extended to create the specified record.

BASIC Data Files are *always* organized into records because the record size must be specified when the file is created using the CREATE statement. Once a BASIC Data File is created, its record size can never be changed. Files of other types can be given a different record size when the file is opened using the OPEN statement as discussed above.

Remember, text files are organized as variable length lines of characters each ending with the return character. Thus, it generally makes sense to access a text file randomly if it is known that each line is exactly the same length and equal to the record size. Text files which contain variable length lines and are accessed randomly with the INPUT# statement will most certainly read partial lines.

Accessing Text Files

A text file is a special type of file which contains ASCII characters organized as *lines*. A line is a sequence of up to 255 characters ending with the Return character (ASCII 13). A text file is created with the TML BASIC CREATE statement where the FILTYP= argument is the value TXT.

TML BASIC provides two statements for accessing text (disk) files and character (device) files: INPUT# and PRINT#. These statements are only with text files. If a program uses these statements for other filetypes, TML BASIC reports the error "File Type Error".

INPUT# Statement

The INPUT# statement reads a *line* of text from a file into an input buffer and then processes the input text according to the list of input variables in its argument list. If the INPUT# statement does not encounter a return character after reading 255 characters, it terminates reading the file, appends a return character to the input buffer, and processes the characters as a single line.

The following is the syntax of the INPUT# statement:

```
INPUT# FileNumber [, RecordNumber ] [; VariableName {, VariableName }]
```

The reserved word INPUT# is followed by the file reference number of an open file, a semicolon, and then a list of variables separated by commas. The following is an example of the INPUT# statement which reads a line into a string variable:

```
INPUT #10; aLine$
```

This form of the INPUT# statement performs sequential access, reading a line of text beginning at the *current file position*. To perform random access using the INPUT# statement, include a record number after the file reference number. Recall that the file must be opened using the OPEN statement with the optional record size argument specified in order to define the size of a record for the text file. Consider the following statements:

```
OPEN "AFILE", AS #10, 15
INPUT #10,4; aLine$
```

The OPEN statement opens the file AFILE with the record size defined as 15 bytes. The INPUT# statement then reads a line of text beginning at the fourth record in the file. The file position for the fourth record is computed with the equation (*RecordNumber* - 1) * *RecordSize*. Thus, the new current file position for the INPUT# statement is calculated as:

$$(4-1) * 15 = 45$$

Therefore, TML BASIC positions the file at the 45th byte of the file before reading the line. Recall that both record numbers and bytes are counted from zero.

The INPUT# statement may contain both string and numeric variables. If a numeric variable is used in an INPUT# statement, TML BASIC automatically converts the string representation of a number into the appropriate numeric type (similar to the VAL statement). When a numeric variable is used in an INPUT# statement and the input line does not contain a string which represents a legal numeric value a "Type Mismatch Error" occurs. If there is not enough data in the input line, the file is read again until all of the variables have been given values.

PRINT# Statement

The PRINT# statement writes a line of text to a file in the same way the PRINT statement does to the screen. The following is the syntax for the PRINT# statement:

```
PRINT# FileNumber [, RecordNumber ] [; Expression {,|; Expression }]
```

The reserved word PRINT# is followed by the file reference number of an open file, a semicolon, and then a list of expressions separated by commas or semicolons. The following is an example of the PRINT# statement which writes several variables to a file:

```
PRINT #10; anInt%, aReal, aStr$
```

PRINT# automatically performs any necessary numeric-to-string type conversions before writing to the file. Numeric values are formatted using the same rules as the PRINT statement. That is, SHOWDIGITS controls the format of numbers generated by PRINT#. Using the comma as the separator between expressions performs a tab to the next print zone before writing the expression, while the semicolon does not. The SPC and TAB functions can be used as well.

This form of the PRINT# statement performs sequential access, writing a line of text beginning at the *current file position*. To perform random access using the PRINT# statement, include a record number after the file reference number. Recall that the file must be opened using the OPEN statement with the optional record size argument specified to define the size of a record in the text file. Consider the following statements:

```
OPEN "AFILE", AS #10, 20
PRINT #10,6; aLine$
```

The OPEN statement opens the file AFILE with the record size defined as 20 bytes. The PRINT# statement then writes a line of text beginning at the sixth record in the file. The PRINT# statement begins at the beginning of the specified random record, and writes the entire value of each expression in its argument list without regard for record size or boundaries. This behavior is unlike that of the WRITE# statement discussed in the next section.

Accessing BASIC Data Files

A BASIC Data File is a special binary coded filetype which provides much faster file access than text files. BASIC Data Files are also called BDF or DATA files. BDF files are faster than standard text files because no text to binary translation is necessary when reading or writing a file. BDF files store data using the same binary representation as the values stored in memory.

TML BASIC provides two statements for accessing BDF files: READ# and WRITE#.

Stucture of a BDF File

BASIC Data files are stored in a special file structure format. Unlike other filetypes, BDF files are *always* organized as fixed size records, regardless of whether or not the file is accessed sequentially or randomly. The record size for a BDF file is specified when the file is created and cannot be changed. If the record size is not specified in the CREATE statement, a default size of 512 bytes is used. When a record size is specified in an OPEN statement for a BDF file, the value is ignored and the record size specified when the file was created is used. The record size for a BDF file is stored as the file's ProDOS 16 filetype.

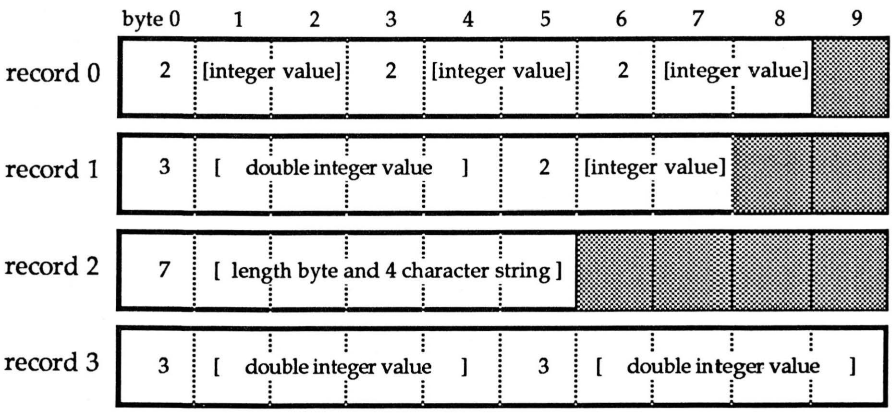
As mentioned above, the data values stored in a BDF file are in the same binary format as the values stored in memory. To identify the type of a value, BDF files also store a *tag byte* immediately preceding a value which uniquely defines the data's type. Table 12-2 shows the values and meaning of each BDF tag byte. The table also shows the number of bytes required to store a value of the given type in a BDF file.

Table 9-2
BDF Tag Byte Values

Tag Byte	Meaning	Bytes in BDF file
0	End-of-file	1
1	*not used*	
2	Integer	3
3	Double Integer	5
4	Long Integer	9
5	Single-precision Real	5
6	Double-precision Real	9
7	String	2 + characters in string

Each data value and its tag byte together are called a *field*. All of the bytes for a field must fit entirely within a record; a field may not span a record boundary. If insufficient space is left in a record to write a field, the field is written to the next record. If the field does not fit within any record, an error occurs.

The following diagram illustrates how various data values might be written to a BDF file containing four records whose record size is 10 bytes. The integer values in the records are tag bytes followed by a description of the data value in brackets. The bytes in each record shown in grey indicate unused storage in the file because the next field in the file could not fit in the record.



If a program attempted to write a string value to this hypothetical file whose size was greater than 10 bytes, an error would occur since the value would not fit within a single 10 byte record.

READ# Statement

The READ# statement reads information from a BDF file into one or more variables. The following is the general syntax for the READ# statement:

```
READ# FileNumber [, RecordNumber ] [; VariableName {, VariableName }]
```

The reserved word READ# is followed by the file reference number of an open BDF file, a semicolon, and then a list of variables separated by commas. The following is an example of the READ# statement which reads three integers:

```
READ #10; anInt1%, anInt2%, anInt3%
```

This form of the READ# statement performs a sequential access, reading one field after the other from successive records in the file. If the values stored at the current file position in the BDF file are not integer values (tag byte not equal to 2), the value must be converted.

If a READ# statement contains a numeric variable, the value at the current file position in the BDF file must also be a numeric value. If the file contains a string value, the "Type Mismatch Error" occurs. If the file does contain a numeric value, but its type does not match the variable in the READ# statement, the value is converted using the same rules as the CONV functions. Thus, it is possible that the conversion will lose precision or even cause an "Overflow Error". If the READ# statement contains a string variable, the value at the current file position must be a string value, otherwise a "Type Mismatch Error" occurs.

An optional form of the READ# statement permits random access to a BDF file. To perform random access using the READ# statement, include a record number after the file reference number. Consider the following statement:

```
    READ #10,3; aStr$, aDblInt@
```

This READ# statement reads a string value beginning at the third record in the file, and then a double integer value. Recall that both record numbers and bytes are counted from zero.

WRITE# Statement

The WRITE# statement writes information to a BDF file. The following is the syntax for the WRITE# statement:

```
    WRITE# FileNumber [, RecordNumber ] [; Expression {,|; Expression }]
```

The reserved word WRITE# is followed by the file reference number of an open file, a semicolon, and then a list of expressions separated by commas or semicolons. The following is an example of the WRITE# statement which writes several variables to a file:

```
    WRITE #10; anInt%, aReal, aStr$
```

This form of the WRITE# statement performs sequential access, writing each successive value at the *current file position*. Each expression in the WRITE# argument list causes a *field* to be written to the BDF file. Recall that a field is a tag byte followed by the binary representation of the value. If a record does not contain enough room to hold all the fields being written to it, the extra fields are written to the next record. If a field cannot fit in any record (it is larger than the record size), an error occurs.

An optional form of the WRITE# statement permits random access to a BDF file. To perform random access using the WRITE# statement, include a record number after the file reference number. Consider the following variation of the above statement:

```
WRITE #10,6; anInt%, aReal, aStr$
```

Accessing Binary Files

A binary file is simply a file consisting of a sequence of bytes without any particular organization or structure such as the BDF files or text files. When a binary file is accessed, the specified number of bytes at the current file position are transferred without any translation into a *structure* array. Any filetype can be opened and accessed as a binary file, including BDF and text files. One important use of binary files in TML BASIC is the reading and writing graphics files.

The TML BASIC GET# and PUT# statements implement access to binary files. By default, these statements transfer a number of bytes equal to the record size of the opened file being accessed. Thus, a file which is accessed using these statements should specify the optional record size argument in the OPEN statement.

GET# Statement

The GET# statement reads a number of bytes from a binary file into a structure array. The following is the syntax of the GET# statement:

```
GET# FileNumber [, [Length] [, RecordNumber]]; StructureVariableReference
```

The reserved word GET# is followed by the file reference number of an open binary file, a semicolon, and then a structure array variable reference (includes a subscript). The number of bytes transferred is equal to the record size of the file. The following statements illustrate the use of the GET# statement:

```
DIM myData!(99)
OPEN "SomeFile", AS #10, 100
GET #10; myData!(0)
```

The DIM statement declares a structure array with 100 elements, thus occupying 100 bytes of storage. The OPEN statement opens a binary file whose record size is 100 bytes and sets the current file position to the first record of the file. Finally, the GET# statement reads the first record (100 bytes) of the file into the structure array beginning at index position 0. Note that the OPEN statement is solely responsible for determining the number of bytes transferred by the GET# statement by defining the record size.

Using the optional *Length* argument in the GET# statement enables a program to override the number of bytes transferred to some value other than the record size. However, the override length value must be less than or equal to the record size. For example, the following GET# statement only transfers 50 bytes from the file.

```
GET #10,50; myData!(0)
```

Each of the above forms of the GET# statement performs sequential access to the binary file. The GET# statement can also be used for random access using the optional *RecordNumber* argument. For example, the following statement reads the second 100 byte record (record numbers begin at 0) from the binary file:

```
GET #10,,1; myData!(0)
```

And the following statement reads only 50 bytes from the second 100 byte record:

```
GET #10,50,1; myData!(0)
```

PUT# Statement

The PUT# statement writes a number of bytes from a structure array to a binary file. The following is the syntax of the PUT# statement:

```
PUT# FileNumber [, [Length] [, RecordNumber]]; StructureVariableReference
```

The reserved word PUT# is followed by the file reference number of an open binary file, a semicolon, and then a structure array variable reference (includes a subscript). The number of bytes transferred is equal to the record size of the file. The following statements illustrate the use of the PUT# statement:

```
DIM myData!(99)
FOR i% = 0 to 99
   myData!(i%) = i%
NEXT i%
OPEN "SomeFile", AS #10, 100
PUT #10; myData!(0)
```

The DIM statement declares a structure array with 100 elements, thus occupying 100 bytes of storage. The structure array is initialized with the FOR loop. The OPEN statement opens a binary file whose record size is 100 bytes and sets the current file position to the first record of the file. Finally, the PUT# statement writes the contents of the structure array to the first record (100 bytes) of the file. Note that the OPEN statement is solely responsible for determining the number of bytes transferred by the PUT# statement by defining the record size.

Using the optional *Length* argument in the PUT# statement, it is possible to override the number of bytes transferred to some value other than the record size.

However, the override length value must be less than or equal to the record size. For example, the following PUT# statement only transfers 50 bytes to the file.

```
PUT #10,50; myData!(0)
```

Each of the above forms of the PUT# statement performs sequential access to the binary file. The PUT# statement can also be used for random access using the optional *RecordNumber* argument. For example, the following statement writes to the second 100 byte record (record numbers begin at 0) from the binary file:

```
PUT #10,,1; myData!(0)
```

And the following statement writes only 50 bytes to the second 100 byte record:

```
PUT #10,50,1; myData!(0)
```

Other File Operations

In addition to the file operations discussed thus far, TML BASIC offers several other statements and functions related to the manipulation of files. The most significant are those statements which relate to detecting and handling an end of file condition. These and other file handling statements and functions are discussed below.

ON EOF# Statement

The ON EOF# statement allows a program to specify what actions to take when a file input statement such as INPUT# or READ# attempts to read past the *end-of-file* (EOF) mark of a file. The ON EOF# statement has a single argument, a file reference number, followed by a sequence of one or more statements. For example:

```
ON EOF #10 PRINT "End of file": CLOSE #10: END
```

When the normal execution of a program encounters an ON EOF# statement, it records that the file associated with the given file reference number has an active ON EOF# statement. The statements on the same line after the ON EOF# statement are not executed.

When a file's end-of-file mark has been reached, there is no more data in the file for an input statement to read, thus the input statement cannot return any value. Without the ON EOF# statement, the program would abort execution, returning the error "Out of Data Error". However, if an ON EOF# statement has been executed for the file whose file reference number matches that in the ON EOF# statement, control automatically transfers to the statements after the ON EOF# statement.

The following program shows how the ON EOF# statement can be used in a very simple program.

```
OPEN "Test", AS #10
ON EOF #10 CLOSE #10: END
ReadAgain: INPUT #10; aLine$
           PRINT aLine$
           GOTO ReadAgain
```

The program opens a text file, executes the ON EOF# statement and then proceeds to read one line at a time from the file and print it to the screen. When the end-of-file is encountered, TML BASIC automatically transfers control to the statements after the ON EOF# statement. In this example, the statements close the open file and then terminates execution of the program.

OFF EOF# Statement

The OFF EOF# statement cancels the effect of the ON EOF# statement. After an OFF EOF# statement has been executed for a file reference number, reading past that file's end-of-file will cause TML BASIC to abort execution of the program and report the error message "Out of Data Error".

EOF Reserved Variable

When TML BASIC encounters an end-of-file, it assigns its file reference number to the reserved variable EOF. The EOF reserved variable can then be used in the code which handles the end-of-file condition for one or more files to determine exactly which file has reached its end-of-file mark. The following is a simple example of using the EOF reserved variable:

```
ON EOF # 10 GOTO HandleEOF
ON EOF # 15 GOTO HandleEOF
ON EOF # 17 GOTO HandleEOF
• • •
HandleEOF: PRINT "End of file encountered for file #"; EOF
           CLOSE #EOF
• • •
```

EOFMARK Function

The EOFMARK function is used to determine the exact location of the end-of-file mark for an open file. The function has a single parameter which is a file reference number of an open file. If the file is not open, an error results. The following example shows how to use the EOFMARK function:

```
FileSize@ = EOFMARK(10)
```

EOFMARK can only be used with disk files. Character device files such as a printer or modem cannot have an end-of-file mark.

FILE Function

The FILE function is used to determine if a file exists as a disk file. The FILE function has a string parameter which specifies the pathname of the file to test for existence. If the file does exist, the FILE function returns a value of one (true), otherwise it returns a value of zero (false). The following example demonstrates how the FILE function might be used.

```
FileOk = 0
DO
   INPUT "Enter a file to open: "; theFilename$
   IF FILE(theFilename$)
     THEN FileOk = 1
     ELSE PRINT "Sorry, that file does not exist"
UNTIL FileOk
OPEN theFilename$, AS #10
```

The FILE function can also have an optional second parameter which specifies a filetype. If the second parameter is given, the FILE function not only checks for the file's existence, but also that its filetype matches the filetype specifed in the second parameter. The second parameter uses the FILTYP= reserved word as described previously with the CREATE statement.

FILTYP Function

The FILTYP function is used to obtain the filetype of an open file. The function has a single parameter which must be a file reference number for an open file. The function returns a integer which is the file's type.

TYP Function

The TYP function is only used with BASIC Data Files. This function examines the type of the next value to be read from an open BASIC Data File and returns an integer which is the tag byte of the next value in the file. The function has a single parameter which is the file reference number of an open file. Table 9-2 defines the tag byte values.

REC Function

The REC function is used with random access files to obtain the current record number of a file. The function has a single parameter which must be a file reference number for an open file. The function returns a double integer which is the record number corresponding to the file's current position. The following is a simple example of the REC function:

```
CurrentRecordPos@ = REC(10)
```

Summary

TML BASIC provides an extensive collection of statements and functions for manipulating files. This chapter has provided an overview of file concepts as related to ProDOS 16, the operating system of the Apple IIGS, as well as a review of the individual TML BASIC statements, functions and reserved variables which provide for file manipulation in TML BASIC programs. You should also reference chapter 10 for more information regarding each of the statements and functions discussed in this chapter.

Chapter 10

Statements and Functions

The TML BASIC language has nearly 200 statements, functions and reserved variables. This chapter serves as a complete reference for each of these language elements with each appearing on its own page.

Statements are the fundamental building block of TML BASIC programs. The source code for a program consists of one or more statements, each appearing on a separate line or on the same line separated by colons. For example:

```
LET Average = (Val1 + Val2 + Val3) / 3
CALL MoveTo(30,20)
```

Predefined functions perform a calculation and return a single value. Therefore, functions are used in expressions. Most of the predefined functions have at least one or more parameters, although some have no parameters. For example:

```
x = SIN(angle)
Message$ = RIGHT$(Message$,5)
Paddle% = PDL9
```

Finally, *reserved variables* are special predefined variables which control or return special system values. Some reserved variables can be assigned values. These are called modifiable reserved variables.

```
theDate$ = DATE$
HPOS = 17
```

For more information about these language elements, see Chapter 7.

The description of each statement, function and reserved variable includes a definition of the syntax for using the language element, a discussion of what action it performs along with a description of its arguments and/or parameters, restrictions and error conditions. Also given is an example of how the language element might be used in a program. Where appropriate, references to other language elements are given to help you better understand its use, and in the case where TML BASIC differs from GS BASIC, a discussion of those differences is provided.

The Syntax Notation

The syntax notation used in this chapter is the same notation described in Chapter 7. The following is an example of the syntax notation used to define a procedure call.

```
    PROC Procedurename  [ ( Expression  { , Expression } ) ]
```

Words which appear in all capital letters denote TML BASIC reserved words and must be used exactly as shown. In the example above, PROC is a reserved word which must appear exactly as shown.

Brackets ([ ]) indicate that the elements between the matching left and right brackets may optionally appear in the syntax. Braces ({ }) indicate that the elements between the matching left and right braces may appear zero or more times in the syntax. The example above, indicates that a procedure's parameter list is optional since it appears in brackets. If a parameter list appears, it may have one or more *Expression* parameters separated by commas as indicated by the braces.

A vertical bar (|) is used to indicate an option. When two or more syntactic elements are separated by a vertical bar, any one of the elements may appear in the syntax, but only one. The vertical bar is not used in the example above.

Special symbols other than braces ({ }), brackets ([ ]) and the vertical bar (|), have special meaning to the syntax for the statement or function being defined, and must appear exactly as shown. For example, the parentheses and commas shown in the example above must appear exactly as shown.

Italicized words indicate that the word is to be substituted with a specific TML BASIC language construct. The italicized word is chosen to help imply the language construct it represents. For example, *Procedurename* is meant to imply that a legal procedure name should appear in its place. Whenever an italicized word appears in the syntax definition, the accompanying text defines the exact meaning of the word. Throughout the syntax, four general italicized words are used: *Expression, NumericExpression, StringExpression* and *Pathname.*

Expression means that any legal TML BASIC expression is to be used in its place. Expressions are constants, variables, functions and operators which evaluate to any type. Expressions are described in Chapter 7. Sometimes the word *Expression* is qualified as either *NumericExpression* or *StringExpression.* In this case, the type of the expression is required to be one of the numeric types or the string type respectively.

The word *Pathname* means that a legal ProDOS 16 pathname must appear. This word is used only in those statements and functions which implement ProDOS 16 operations. *Pathnames* are described in Chapter 9.

ABS Function

Syntax

```
    ABS(NumericExpression)
```

Action

The ABS function returns the absolute value of the *NumericExpression*. The *NumericExpression* may be any TML BASIC numeric type, and ABS returns a value which is the same type as *NumericExpression*.

The absolute value of a numeric expression is its magnitude without regard to its sign. For example, the absolute value of -12 is 12; and the absolute value of +12 is 12. The absolute value of zero is zero.

Example

```
A = -438
PRINT  ABS(438)
PRINT  ABS(A)
PRINT  ABS(-34.92)
```

OUTPUT:

```
438
438
34.92
```

ANU Function

Syntax

`ANU(Rate, Periods)`

Action

The annuity function, computes the annuity, ANU(*Rate, Periods*) that is equal to the following calculation:

(1 - (1 + *Rate*) ^ (-*Periods*)) / *Rate*

where *Rate* and *Periods* can be any numeric type. *Rate* indicates the interest rate and *Periods* represents the number of time periods for which to compound the interest.

The calculation ANU(*Rate, Periods*) is more accurate than the straightforward computation of the expression shown above using normal arithmetic and exponentiation operations. The annuity function is directly applicable to the computation of present value and future value of ordinary annuities.

An annuity is a series of equal payments made at regular intervals with interest compounded at a certain rate. The number of payments is always one more than the number of time periods. Present value can be calculated using the annuity function alone but future value is calculated with the annuity function and also the compound function.

See Also

COMPI

Example

```
PRINT ANU(0.08,180)

Amount  = 10000         ' Initial investment amount
Rate    = 0.08          ' Interest rate
Periods = 4             ' Number of time periods

PRINT "Present Value = "; Amount * ANU(Rate, Periods)
PRINT "Future Value  = "; Amount * COMPI(Rate,Periods) * ANU(Rate,Periods)
```

OUTPUT:

```
1.25
Present Value = 33121.27
Future Value  = 45061.12
```

ASC Function

Syntax

```
    ASC(StringExpression)
```

Action

The ASC function returns an integer value which is the ASCII (American Standard Code for Information Interchange) character code for the first character of *StringExpression*. If the value of *StringExpression* is a null string then the result is -1.

To convert an integer which represents an ASCII character code into a string, use the function CHR$, which creates a single character string from the given character code.

See Also

CHR$
Appendix E

Example

```
S$ = "hello"
PRINT ASC(S$)
PRINT ASC("TML BASIC")
PRINT ASC("")
```

OUTPUT:

```
104
84
-1
```

ASSIGN Statement

Syntax

```
    ASSIGN DeviceName, SlotNumber [,AUTO]
```

Action

The ASSIGN statement associates a character device with a slot or port number. *DeviceName* is a string expression beginning with a period, followed by a letter (A-Z, a-z) followed by zero or more letters or digits, that indicates a filename (case is not significant). The *SlotNumber* argument is an integer value in the range -1 to 7. The optional AUTO argument indicates that TML BASIC should also send a line-feed after each carriage-return sent to the device.

After a device name has been defined it can be used in the OPEN statement as a character device filename. The device can then be accessed as a file using TML BASIC's file I/O statements.

TML BASIC allows up to 12 device names to be defined (including the six predefined names). The device names are stored in an internal *device table.* A value of 1 through 7 defines the slot number of the character device. A value of zero (0) defines a null device, and the value -1 deletes a device name from the current device table.

TML BASIC predefines six character device names. These names can be deleted if needed. The following table lists the six predefined device names.

DeviceName	Slot	Auto Line-feed	Description
.CONSOLE	3	Off	C3COUT1
.PRINTER	1	On	
.MODEM	2	Off	
.MEMBUFR	-	Off	Pseudo device (255 byte buffer)
.NETPTR1	7	On	AppleTalk printer driver
.NULL	0	Off	A bit bucket, read=CR

See Also

OPEN

Example

```
ASSIGN ".MYPLOTTER", 6     'Define the device .MYPLOTTER at slot 6
```

ATN Function

Syntax

```
    ATN(NumericExpression)
```

Action

The ATN function returns, in radians, the trigonometric arctangent (inverse tangent) of *NumericExpression*. In other words, ATN returns the angle whose tangent is *NumericExpression*.

The value returned represents an angle in the range -π/2 to +π/2 radians.

See Also

COS
PI
SIN
TAN

Example

```
PI# = ATN(1.0) * 4      ' Calculate the value of PI using ATN
PRINT PI#
```

OUTPUT:

```
3.141593
```

AUXID@ Reserved Variable

Syntax

```
    AUXID@
```

Action

The AUXID@ reserved variable is set each time an OPEN or FILE statement is executed. It returns a double integer which is the subtype of the file specified in the last executed OPEN or FILE statement.

See Also

OPEN
FILE

Example

```
Exists% = FILE("/TML/TMLBASIC")
PRINT Exists%, AUXID@
```

OUTPUT:

```
1          0
```

BREAK ON, BREAK OFF Statements

Syntax

```
BREAK ON
BREAK OFF
```

Action

During normal program execution, TML BASIC programs monitor the keyboard for a Control-C keypress. If a Control-C is typed, program execution aborts and returns control to TML BASIC (for Compile to Memory) or to the Apple IIGS Finder (for Compile to Disk). TML BASIC only monitors the keyboard *between* statements. Thus, it is not possible to abort using Control-C while an INPUT statement is waiting for user input.

Control-C monitoring can be suppressed by the BREAK OFF statement and re-enabled with the BREAK ON statement. If a Control-C is typed while BREAK OFF is active, it is treated like any other character and program execution continues normally. All programs begin with BREAK ON enabled.

Because TML BASIC must generate code between each statement in a program to check for the Control-C keypress, programs are larger and slower than if the Control-C is not checked. TML BASIC allows programs to turn off this code generation entirely using the $KeyboardBreak metastatement or by turning off the Keyboard Break option in the Preferences Dialog. If Control-C code generation checking is turned off, programs will run faster and be smaller, however, it will be impossible to abort execution of the program using Control-C regardless if BREAK ON is active.

See Also

ON BREAK
Chapter 6, Preferences Dialog
Appendix B

Example

```
•
•
BREAK OFF          'Turn off Control-C checking while updating screen
GOSUB UpdateScreen
BREAK ON           'Restore Control-C checking
•
•
```

BTN Function

Syntax

```
    BTN(ButtonNumber)
```

Action

The BTN function returns the state of the three Apple IIGS sense inputs. *ButtonNumber* must be an integer in the range 0 to 2. Any number outside this range will produce an "Illegal Quantity Error".

BTN returns the integer values 0 or 1 reflecting the state of the input. Various devices can control the state of these inputs including the buttons on paddles or joysticks, and the Open-Apple and Option keys.

The following shows the legal values for *ButtonNumber* and the input it tests.

Command	Input Address	Explanation
BTN(0)	$E0C061	returns 1 if Open-Apple key is down, 0 if up
BTN(1)	$E0C062	returns 1 if option key is down, 0 if up
BTN(2)	$E0C063	

Example

```
IF BTN(0)=1 THEN PRINT "Open Apple Down" ELSE PRINT "Open Apple Up"
IF BTN(1)=1 THEN PRINT "Option Key Down" ELSE PRINT "Option Key Up"
```

CALL Statement

Syntax

```
    CALL ToolboxName [ ( Expression {, Expression } ) ]
    _ToolboxName [ ( Expression {, Expression } ) ]
```

Action

The CALL statement executes a named procedure or function in an Apple IIGS toolset. The declarations for Toolbox procedures and functions are defined in the several predefined libraries shipped with TML BASIC in the folder LIBRARIES. See Appendix C for a complete list of the Toolbox libraries and the procedures and functions declared in them.

Following the reserved word CALL is the name of the toolbox procedure. If the procedure has parameters, they are given after the toolbox name enclosed in parentheses. The rules for matching parameters are the same as for normal BASIC procedures. If the Toolbox routine is a function then its return values are placed in the *CALL return stack* See the description of the reserved variable R.STACK for a description of the Call return stack.

In order to call a Toolbox procedure, the library containing the declaration of the routine must appear in a LIBRARY statement, otherwise TML BASIC reports the error "Toolbox procedure *xxx* is not defined", where *xxx* is the name of the procedure.

TML BASIC allows the use of the underscore character (_) as a shorthand form of the CALL reserved word. Any time a CALL statement is used, it can be substituted with the underscore character. See the example below.

Chapter 11 provides a detailed discussion of the Apple IIGS Toolbox and how to access it from TML BASIC.

See Also

CALL%
R.STACK
Chapter 11
Appendix C

Example

```
LIBRARY "QuickDraw"    'Load the QuickDraw library
CALL MoveTo(10,23)     'Call the MoveTo procedure in the QuickDraw library
_MoveTo(10,23)
```

CALL% Statement

Syntax

```
    CALL% FunctionNumber, ToolSetNumber, ResultSize
        [ ( Expression {, Expression } ) ]
```

Action

The CALL% statement is a variation of the CALL statement for calling the Apple IIGS Toolbox procedures and functions. The CALL% statement allows a program to call a Toolbox procedure by specifying its *FunctionNumber, ToolSetNumber* and function *ResultSize;* while the CALL statement calls a Toolbox procedure by its name.

As described in Chapter 11, section "The Toolbox Libraries", the Apple IIGS Toolbox is divided into a collection of individual toolsets, each assigned a unique *Tool set number*. Furthermore, each procedure and function within a toolset is assigned a unique *function number*. Together, these two numbers uniquely identify every procedure and function in the Toolbox. It is these two numbers that are used in the CALL% statement to call a Toolbox routine. Appendix C lists each of the procedures and functions in the Toolbox with their tool set and function numbers.

If the procedure has parameters, they are given after the toolbox name enclosed in parentheses. The rules for matching parameters are the same as for normal BASIC procedures.

See Also

CALL
R.STACK
Chapter 11
Appendix C

Example

```
CALL% 58,4,0 (10,23)    'Call the MoveTo procedure in the QuickDraw library

LIBRARY "QuickDraw"    'Load the QuickDraw library
CALL MoveTo(10,23)     'Equivalent to the CALL% statement
```

CATALOG Statement

Syntax

```
    CATALOG [StringExpression]
    CAT [StringExpression]
```

Action

CATALOG or CAT displays a listing of the disk contents of the current directory. The CAT statement only displays a subset of the complete information displayed by the CATALOG statement. The current directory is the ProDOS 16 prefix 0 which is also the value of the reserved variable PREFIX$.

If the optional *StringExpression* argument appears, the contents of the directory indicated by *StringExpression* is displayed. If the value of *StringExpression* does not represent a valid ProDOS 16 pathname of a directory file, the error "Path Not Found" occurs.

If OUTPUT# is set to anything other than 0, the directory listing will be sent to the specified OUTPUT# file and not to the screen.

See Also

OUTPUT#
PREFIX$
Chapter 9, Pathnames

Example

```
CATALOG "/TML/PART1.EXAMPLES"
CAT "/TML"
```

CHAIN Statement

Syntax

```
    CHAIN PathName
```

Action

The CHAIN statement is used to launch another ProDOS 16 application from a TML BASIC program. When the CHAINed application quits, control returns to the TML BASIC program at the statement immediately after the CHAIN statement. When control returns, all open files remain open and all variables remain in tact. The *PathName* argument is a string expression which must represent a legal pathname to a ProDOS 16 application.

Compiler/Interpreter Differences

TML BASIC can only chain to compiled ProDOS 16 applications, while GS BASIC chains control to GS BASIC source code programs. Because GS BASIC chains to source code, an optional line number or label may be specified as the location to begin execution in the program. TML BASIC only transfers control to the beginning of an application.

Example

```
CHAIN "PAYROLL"

PART2$ = "/ACCOUNTING/TAXPROGRAM"
CHAIN PART2$
```

CHR$ Function

Syntax

```
    CHR$(NumericExpression)
```

Action

The CHR$ function returns a one character string whose single character has the ASCII code which is *NumericExpression*. The value of *NumericExpression* must be in the range 0 to 255 inclusive. If the value is outside this range then the error "Illegal Quantity Error" occurs. Real values passed will automatically be rounded to the nearest integer.

The CHR$ function complements the ASC function, which returns the ASCII code of the first character of a string.

See Also

ASC
Appendix E

Example

```
PRINT CHR$(65)
PRINT CHR$(34);"HELLO"; CHR$(34)   ' CHR$(34) is double quote character
```

OUTPUT:

```
A
"HELLO"
```

CLEAR Statement

Syntax

```
    CLEAR
```

Action

The CLEAR statement is used to set all numeric variables to zero, string variables to null, and closes all open files. Note that if CLEAR is used inside a loop, the loop counter is cleared causing an infinite loop.

The ERASE statement should be selectively free storage for arrays.

Compiler/Interpreter Differences

Unlike GS BASIC, TML BASIC does not support dynamically setting the stack and data segments. Thus, the CLEAR statement in TML BASIC does not support any arguments for specifying the new size of a data segment.

See Also

ERASE

Example

```
DIM StrArray$(1)

StrArray$(0) = "TML BASIC"
StrArray$(1) = "TML Pascal"

PRINT "*"; StrArray$(0); "*"; StrArray$(1); "*"
CLEAR
PRINT "*"; StrArray$(0); "*"; StrArray$(1); "*"
```

OUTPUT:

```
*TML BASIC*TML Pascal*
***
```

CLOSE and CLOSE# Statements

Syntax

```
    CLOSE [# FileNumber ]
```

Action

The CLOSE and CLOSE# statements are used to close files that were previously opened with an OPEN statement. CLOSE# closes the file whose file reference number is equal to *FileNumber*. The *FileNumber* parameter is an integer in the range 0 to 31. If *FileNumber* is outside this range, or if no open files have the specified file number, the "File Not Open Error" occurs.

Before ending program execution, all open files should be closed using the CLOSE# or CLOSE statements. Any files closed during program execution must be reopened before they can be accessed again.

CLOSE closes ALL files that are open when the statement is executed. In addition, TML BASIC closes all open files when the RUN and END statements are executed and when a program terminates. Unlike the RUN statement, the CHAIN statement does not cause any files to be closed.

Example

```
CLOSE  #4       'Close file previously opened as file number 4.
CLOSE           'Close all open files
```

COMPI Function

Syntax

```
    COMPI(Rate, Periods)
```

Action

The compound interest function, COMPI(*Rate, Periods*), computes the expression:

$(1 + Rate)$ ^ $Periods$.

where *Rate* and *Periods* can be any numeric type. *Rate* indicates the interest rate and *Periods* represents the number of periods for which interest in calculated.

When the rate is small, COMPI(*Rate, Periods*) gives a more accurate result for the computation than does the straightforward computation of (1+*Rate*)^*Periods* by addition and exponentiation. COMPI is directly applicable to computation of present and future values.

See Also

ANU

Example

```
Rate    = 0.08   'Interest rate is 8%
Periods = 10     'Investing for 10 years compounding annually
Amount  = 10000  'Principle to invest is $10,000

PRINT COMPI(Rate,Periods) * Amount
```

OUTPUT:

```
21589.25
```

CONV Functions

Syntax

```
    CONV [#|%|@|&|$] (AnyExpression)
```

Action

The CONV functions are a set of generalized conversion functions which convert any numeric or string expression into a value of the specified type. The type character used with the CONV function indicates the result type of the function.

If a numeric expression evaluates outside of the specified result type, an "Overflow Error" occurs. If a string expression is converted to a numeric type, the string value must represent a legal numeric string, otherwise, the value zero (0) is returned. When *AnyExpression* is a string expression, the effect is the same as the VAL function.

See Also

VAL
Chapter 7

Example

```
PRINT "My address is " + CONV$(12*4) + " Memory Lane"
myReal = 43.21
PRINT CONV%(myReal)
PRINT CONV%(60000)      'This statement causes an Overflow Error
```

OUTPUT:

```
My address is 48 Memory Lane
43
```

COS Function

Syntax

```
    COS(NumericExpression)
```

Action

Returns the trigonometric cosine of *NumericExpression*. *NumericExpression* is an angle expressed in radians. To convert radians to degrees, multiply by 180/π. To convert degrees to radians, multiply by π/180.

See Also

ATN
PI
SIN
TAN

Example

```
PRINT "Cosine of 45 degrees = '; COS(45 * PI/180)
```

OUTPUT:

```
0.7071068
```

CREATE Statement

Syntax

```
    CREATE PathName [,FILTYP= DIR|TXT|SRC|BDF|FileType [,SubType]]
```

Action

The CREATE statement is used to create a disk file. The created file may be a subdirectory, text file, Basic Data File, or any other valid ProDOS 16 filetype.

The *PathName* argument is a string expression which must represent a legal ProDOS 16 filename or pathname. If an invalid *PathName* is given then the "Bad Path Error" occurs. If the CREATE statement attempts to create a file on a disk which is write protected, the "Write Protect Error" occurs.

The FILTYP= argument may optionally appear after the *PathName* argument to specify the filetype of the created file. FILTYP= may specify one of the four predefined filetypes using the filetype's mneumonic name or an arbitrary filetype by specifying an unsigned integer *FileType* value. If the FILTYP= argument does not appear, then the CREATE statement creates a text file (TXT) by default. The table below summarizes the predefined filetype names, their alternate names and meaning.

Filetype Mneumonic	**Alternate Mneumonic**	**Meaning**
DIR	CAT	Subdirectory
TXT	TEXT	Text file
SRC		Source file (a text file)
BDF	DATA	Basic Data File

Appendix F contains a list of the most often used ProDOS 16 filetypes.

If the FILTYP= argument appears, then it may optionally be followed by a file subtype specification. *SubType* is an unsigned integer value in the range 0 to 65,535. If the subtype is not specified the default value of zero is used except for the case of Basic Data Files. If the specified file type is a subdirectory (DIR) then the subtype is zero regardless of the value specified. The meaning of the subtype varies depending upon the file type (See Chapter 9).

Basic Data Files require that the subtype value be in the range 3 through 32,767. The reason for this is that TML BASIC uses the subtype of a Basic Data File as the file's logical record size. The logical record size of a Basic Data File must be known in order to support random-access to the file's records. See Chapter 9 for a discussion of files in TML BASIC.

An attempt to create an already-existing file using the CREATE statement cases the "Duplicate File Error" to occur.

Example

```
CREATE "/TML/MY.EXAMPLES", FILTYP=DIR  'Create a new subdirectory

CREATE "GRADES"                        'Create a text file

CREATE "MYROLODEX",FILTYP=BDF,100      'Create Basic Data File
                                       ' with record size (subtype) of 100

IllegalFileName$ = "()#$%^"
CREATE IllegalFileName$,FILTYP=0       'Causes Bad Path Error
```

DATA Statement

Syntax

```
    DATA constant {, constant}
```

Action

The DATA statement declares constant values for the READ statement. A DATA statement contains one or more constants separated by commas. A constant can be a string constant or any floating-point or integer constant.

A program can have as many DATA statements as required, and they need not be located on successive lines. During execution, READ statements access the DATA constants from left to right and top to bottom in the order in which they appear in the source code. A "Type Mismatch Error" occurs if a READ statement attempts to read a string constant into a numeric variable.

The RESTORE statement is used to reread constants from the first DATA statement in the program or any specified DATA statement. If a program attempts to READ more data than exists in DATA statements, an "Out of Data" error occurs.

See Also

READ
RESTORE

Example

```
READ A$,B$

RESTORE Names
READ C$,D$

PRINT A$,B$,C$,D$
END

Names: DATA "Apple", "Orange"
       DATA "Pear", "Grape"
```

OUTPUT:

```
Apple     Orange     Apple     Orange
```

DATE Function

Syntax

```
    DATE(NumericExpression)
```

Action

The DATE function reads the Apple IIGS clock to return the current date information as an integer rather than a string as returned by the DATE$ function. The value of *NumericExpression* must be in the range 0 through 4 inclusive, otherwise the "Illegal Quantity Error" occurs.

The following table shows the values returned by the DATE function for each legal parameter value.

Function	**Value returned**
DATE(0)	Year - 1900
DATE(1)	Year
DATE(2)	Month, where 1=January, 2=February, ... 12=December
DATE(3)	Day of the month, 1 through 31
DATE(4)	Day of week, 1 through 7 where 1=Sunday

Actually, the Apple IIGS clock is only read when the parameter value is zero. DATE(0) reads the Apple IIGS clock for all date information and then updates the values which will be returned by the other DATE function calls. This feature protects programs from classical "clock rollover" problem.

Example

```
ReadDate%  = DATE(0)    'Read the Apple IIGS Date information
Year%      = DATE(1)
DayOfWeek% = DATE(4)

PRINT "The year is "; Year%
IF (DayOfWeek% = 1) OR (DayOfWeek% = 7) THEN PRINT "This is the weekend!"
```

OUTPUT:

```
The year is 1987
This is the weekend!
```

DATE$ Function
DATE$ Statement

Syntax

```
    DATE$
    DATE$ Year, Month, DayOfMonth
```

Action

DATE$ is both a function and a statement in TML BASIC. The DATE$ statement has three arguments, while the DATE$ function has none.

The DATE$ function reads the Apple IIGS clock and returns the current date as a string. The form of the string depends upon the date format chosen in the Apple IIGS control panel. The date formats are MM/DD/YY, DD/MM/YY, YY/MM/DD, where MM stands for the month, DD stands for the day and YY stands for the year. See your *Apple IIGS Owner's Guide* for information on how to use the Control Panel.

The DATE$ statement is used to change the date settings of the Apple IIGS clock. The year is specified by the *Year* parameter, the month by the *Month* parameter and the day of the month by the *DayOfMonth* parameter. The *Year* parameter is the year minus 1900 and must be in the range 0 through 255 inclusive. For example, 87 indicates the year 1987. *Month* should be in the range 1 through 12 inclusive and *DayOfMonth* should be in the range 1 through 31 inclusive.

Example

```
DATE$ 66,12,6   'Set Apple IIGS clock to December 6, 1966
PRINT DATE$; " was a memorable date."
```

OUTPUT:

```
12/ 6/66 was a memorable date.
```

DEF FN Statement

Syntax

```
DEF FN functionname [%|@|&|#|$] [ ( parameter {, parameter} ) ] = expression

DEF FN functionname [%|@|&|#|$] [ ( parameter {, parameter} ) ]
   LOCAL variable {, variable }
   •
   • statements
   •
   FN functionname [%|@|&|#|$] = expression
   •
END FN [ functionname ]
```

Action

The DEF FN statement is used to define functions. Functions are used to group together one or more statements that compute and return a value. Functions are called using the reserved word FN.

There are two types of functions in TML BASIC: *single-expression (simple) functions* and *multiline functions*. Simple-expression functions are contained on a single line and have only one expression for computing the value of the function. The type of the expression must be compatible with the function name. Multiline functions may contain several statements bracketed by the DEF FN and END FN statements. At least one of the statements must be an assignment statement to the function variable.

The *functionname* in a function declaration declares the name of the function. The name may not be any of the TML BASIC reserved words. Following the function name is an optional type character used to specify the type of the function's result value. If no type character is given the function returns a single-precision real value. A function may optionally have a sequence of parameters whose values are used to compute the value of the function. A parameter may be any numeric or string type, but not an array.

The position of a function declaration in the source of a program has no effect on program flow or when the function can be called. A function declaration behaves like a large comment around all the statements in the function so that a program does not have to direct program flow around the function declaration. In addition, a function can be called anywhere in the program, even if the function is declared later in the source code.

See Chapter 8 for a complete discussion of functions.

See Also

DEF PROC
LOCAL
Chapter 8

Example

```
DEF FN Circumf(X) = X*2*PI

DEF FN Factorial#(n%)
   LOCAL total#
   IF n% < 0 THEN
      FN Factorial# = 1
   ELSE
      FOR i% = n% TO 2 STEP -1
         total# = total# * i%
      NEXT i%
   END IF
END FN Factorial#
```

DEF LIBRARY Statement

Syntax

```
DEF LIBRARY LibraryName
        •
        • statements
        •
END LIBRARY
```

Action

The DEF LIBRARY statement is used to create a *library*. A library is a special source code construct that groups together procedure and function declarations so that they can be compiled separately from any program. Libraries can then be used in other programs just as if the source code in the library textually appeared in the program. A library essentially behaves as a *repository* of code for other programs to use.

The source code for a library must begin with the DEF LIBRARY statement. It must be the first non-empty, non-comment line in the source code. In addition, the source code must end with the END LIBRARY statement. All of the statements between the DEF LIBRARY and the END LIBRARY statements are part of the library. Only five types of statements are allowed in a library: LIBRARY, REM, DIM, DEF PROC and DEF FN. No other statements are allowed in the library (including the DIM DYNAMIC statement). The reason for this is that the code in a library does not create a program that can be executed. It only contains code that other programs can call. Because a library is never executed, it does not make sense for it to contain executable statements.

Of course, the statements within a procedure or function declaration (DEF PROC and DEF FN) may be any legal TML BASIC statements. These statements are executed whenever the procedure or function is called by a program that uses the library.

To use a library in a program, the LIBRARY statement is used. When the LIBRARY statement appears in a program (or even another library), TML BASIC makes all of the declarations in the library available to the program just as if the source code in the library appeared in the program itself.

See Chapter 8 for a complete discussion of libraries.

See Also

LIBRARY
Chapter 8
Chapter 11

Example

```
DEF LIBRARY IntegerStack

   REM This library implements a simple 100 element integer stack.

   DIM theStack%(99)                    'Declare the stack

   DEF PROC ClearStack                  'Make the stack empty
      stackTop% = -1
   END PROC ClearStack

   DEF PROC Push(aValue%)               'Push a new value onto the stack
      IF stackTop% < 99 THEN
         stackTop% = stackTop% + 1
         theStack%(stackTop%) = aValue%
      END IF
   END PROC Push

   DEF FN Pop%                          'Remove the top element of the stack
      IF stackTop% >= 0 THEN
         FN Pop% = theStack%(stackTop%)
         stackTop% = stackTop% - 1
      END IF
   END FN Pop%

END LIBRARY
```

DEF PROC Statement

Syntax

```
DEF FN procedurename [ ( parameter {, parameter} ) ]
   LOCAL variable {, variable }
   •
   • statements
   •
END FN [ procedurename ]
```

Action

The DEF PROC statement is used to define procedures. A procedure is a construct that allows a program to group together related statements. Procedures behave much like subroutines (GOSUB) except they provide additional capabilities not available with subroutines. Procedures are called using the PROC statement.

The *procedurename* in a procedure declaration declares the name of the procedure. The name may not be any of the TML BASIC reserved words. A procedure may optionally have a sequence of parameters whose values are used in the statements within the procedure. A parameter may be any numeric or string type, but not an array.

A procedure may contain the LOCAL statement to declare variables local to the procedure. Procedures may also call themselves recursively.

The position of a procedure declaration in the source of a program has no effect on program flow or when the function can be called. A procedure declaration behaves like a large comment around all the statements in the procedure so that a program does not have to direct program flow around the procedure declaration. In addition, a procedure can be called anywhere in the program, even if the procedure is declared later in the source code.

See Chapter 8 for a complete discussion of procedures.

See Also

DEF FN
LOCAL
Chapter 8

Example

```
REM Test parameter passing for a procedure

anInt% = 1
aDblInt@ = 44932
aLongInt& = -482
aSglReal = 932.8
aDblReal# = 34.238e43
aString$ = "Hello"

PROC TestParams(anInt%,aDblInt@,aLongInt&,aSglReal,aDblReal#,aString$)

DEF PROC TestParams(I%,D@,L&,Sgl,Dbl#,Str$)
   PRINT I%, D@, L&
   PRINT Sgl, Dbl#, Str$
END PROC
```

DELETE Statement

Syntax

```
    DELETE PathName
```

Action

The DELETE statement is used to remove a subdirectory or file from a disk. *PathName* is a string expression that contains the filename or subdirectory to be deleted. *PathName* must be a legal ProDOS 16 pathname, otherwise the "Bad Path Error" occurs.

A subdirectory may be removed only if all files in that subdirectory have been deleted. Even if all files in a root directory have been deleted, the root directory itself cannot be deleted.

See Also

CREATE
Chapter 9

Example

```
DELETE "MYFILE"

SomeFile$ = "/TML/MY.EXAMPLES/XYZ.BAS"
DELETE SomeFile$
```

DIM Statement
DIM DYNAMIC Statement

Syntax

```
DIM ArrayName ( Subscript {, Subscript} )
      {, ArrayName ( Subscript {, Subscript} ) }

DIM DYNAMIC ArrayName ( Subscript {, Subscript} )
             {, ArrayName ( Subscript {, Subscript} ) }
```

Action

The DIM statement is used to declare one or more array variables and their size and number of dimensions. An array is a collection of values of the same type referred to by the same variable name. Each *subscript* in a DIM statement defines the number of elements in that array dimension. The number of elements in a dimension is one greater than the value given. This is because the array elements are referenced from zero. For example,

```
DIM Sales%(11)
```

defines a one-dimensional integer array variable *Sales%*, consisting of 12 elements and numbered 0 through 11. TML BASIC sets each element of a numeric array to zero, and each element of a string array to the null string when the array is created.

Arrays can have one or more dimensions, up to a maximum of eight. The maximum number of elements per dimension is 32,768. The maximum total size of a single array is 64K bytes.

If an array variable is used without a preceding DIM statement, TML BASIC implicitly DIMensions the array. The array is declared with the same number of dimensions as are referenced in the undeclared array, and each dimension is created with 11 elements (numbered 0 through 10).

The DIM statement is used to create *static* sized array variables. Static arrays have a fixed size that may not change during execution of a program. To create an array that can change size during execution or one whose size cannot be determined at compile time, use the DIM DYNAMIC statement.

See Chapter 7 for complete details about arrays in TML BASIC.

See also

UBOUND
Chapter 7

Example

```
DIM MyArray% (15,20,3), YourArray (5,2,9)
DIM QDString!(255)

DIM DYNAMIC Scores@(n%)
```

DO...WHILE...UNTIL Statements

Syntax

```
DO
    •
    • Statements
    •
[WHILE [Expression ]]
    •
    • Statements
    •
UNTIL [Expression ]

WHILE [Expression ]
    •
    • Statements
    •
UNTIL [Expression ]
```

Action

The DO...WHILE...UNTIL statements are used to create powerful looping constructs. Using different combinations of the three reserved words, just about any control structure can be created. UNTIL is used to create loops that repeat *until* an expression evaluates to true. And the reserved word WHILE is used to create loops that repeat *while* an expression remains true.

The expressions used with either the WHILE or the UNTIL statements may be any valid TML BASIC expression. If the expression evaluates to a non-zero value, it is considered as TRUE. If the expression evaluates to the value zero, or a null string, it is considered as FALSE.

The first form of the DO...WHILE...UNTIL statement is the simple DO...UNTIL construct. For example,

```
DO
    •
    • statements
    •
UNTIL Val% = 10
```

In this case, the loop executes the statements between the DO and the UNTIL statements *until* the expression *Val% = 10* becomes true. If the expression never becomes true, the loop repeats indefinitely. This form of the loop may have a WHILE statement added between the DO and the UNTIL statements. In this case, the loop will terminate if the expression after the WHILE statement becomes false.

The second form of the DO...WHILE...UNTIL statements is the WHILE...UNTIL statement. For example,

```
WHILE Val% = 10
    •
    • statements
    •
UNTIL
```

In this example, the loop executes the statements between the WHILE and the UNTIL statements *while* the expression *Val% = 10* remains true. If the expression never becomes true, the loop repeats indefinitely. This form of the loop may also have an expression after the UNTIL, in which case, the loop must satisfy both conditions in order to repeat. For example,

```
WHILE Val% = 10
    •
    • statements
    •
UNTIL anotherVal <> 0
```

Although TML BASIC does not care about the format of source code, it is generally a good idea to indent the source code statements a few spaces to better indicate the statements contained in the loop.

END Statement

Syntax

```
    END
```

Action

END terminates execution of a TML BASIC program. Before the END statement terminates the program, it first closes all open files. TML BASIC automatically inserts an END statement after the last statement in a program so that a program does not "run off the bottom".

If a program was run from within the TML BASIC environment using the **To Memory & Run** compile option, control returns back to TML BASIC after the END statement is executed. If the program was launched from the Apple IIGS Finder, control returns to the Finder after the END statement is executed.

There are other forms of the END statement. In particular, the END FN, END PROC and END LIBRARY statements. See the statements DEF FN, DEF PROC and DEF LIBRARY for more information about these variations of the END statement.

See Also

DEF FN
DEF PROC
DEF LIBRARY
STOP

Example

```
PRINT "This is a very short program"
END
```

OUTPUT:

```
This is a very short program
```

EOF Reserved Variable

Syntax

```
    EOF
```

Action

The EOF reserved variable is assigned the file reference number of the file for which the end of file has most recently been detected. If no file has yet encountered its end of file then EOF contains the value zero.

See Also

ON EOF#
Chapter 9

Example

```
PRINT "End of file was most recently detected for file #"; EOF
```

EOFMARK Function

Syntax

```
EOFMARK(FileNumber)
```

Action

The EOFMARK function returns the current end-of-file mark for the file opened with *FileNumber* as its file reference number. The end-of-file mark indicates the current size of the file on disk. The value returned is a double integer.

See Also

OPEN
Chapter 9

Example

```
OPEN "MYFILE", AS #1
PRINT EOFMARK(1)  'Print the value of the End of file mark for MyFile
```

ERASE Statement

Syntax

```
    ERASE ArrayVariable {, ArrayVariable }
```

Action

The ERASE statement deletes dynamic arrays and resets static arrays. Following the reserved word ERASE are one or more array variable names separated by commas. The array variable names must already be declared before appearing in the ERASE statement, otherwise resulting in an error.

If the named array is a dynamic array, the memory allocated for the array is released making it available for other program needs. If the named array is a static array, the memory allocated to the array cannot be deallocated, however, every element in the array is assigned the value zero or the null string depending upon the element type of the array.

The CLEAR statement can be used to erase all arrays at once.

See Also

CLEAR
DIM

Example

```
PRINT FRE                    'Print available memory

DIM DYNAMIC BigArray(2000)   'Allocate a large dynamic array
BigArray(943) = 123
PRINT FRE                    'Print available memory

ERASE BigArray
PRINT FRE                    'Print available memory
END
```

ERR Reserved Variable

Syntax

```
    ERR
```

Action

When TML BASIC detects a runtime error in a program, it assigns the reserved variable ERR to the number of the error which was detected. ERR is typically used in the sequence of statements after the ON ERR statement. ERR returns an integer value.

Appendix A defines all the runtime errors and their error numbers.

See Also

ON ERR
Appendix A

Example

```
ON ERR GOTO ErrHandler
i% = 20000
i% = i% + 25000
PRINT "i% = "; i%
END

ErrHandler:
   IF ERR = 1 THEN
      i% = 0
      RESUME
   ELSE
      STOP
   END IF
```

OUTPUT:

```
i% = 25000
```

ERROR Statement

Syntax

```
    ERROR ErrorNumber
```

Action

ERROR is used for generating user-defined errors at execution time, which can be trapped by the ON ERR statement. *ErrorNumber* is an integer constant in the range 1 to 255. The reserved variable ERR is assigned the value of *ErrorNumber*.

TML BASIC reserves the error numbers 1 through 127, inclusive, for its own use. Several of these error numbers are currently defined in Appendix A. Error numbers 128 through 255 are available for any user defined meaning.

See Also

ERR
ON ERR
Appendix A

Example

```
ERROR 1         'Equivalent to the TML BASIC Overflow Error
ERROR 128       'User defined error 128
```

EVENTDEF Statement

Syntax

```
    EVENTDEF Index, Label
```

Action

The EVENTDEF statement is used to store subroutine labels in the *Event Dispatch Table*. The event dispatch table is a special data structure defined for directing program control to event-handling subroutines when an event occurs in a desktop application. Events are detected by the TASKPOLL statement.

The event dispatch table has 64 entries numbered 0 through 63. The first 32 entries (0 through 31) are reserved for use with the TASKPOLL statement. The entries correspond directly to the event codes returned by the Window Manager TaskMaster routine which are the events detected by the TASKPOLL statement.

The following table shows the meaning of the TASKPOLL event codes:

TASKPOLL Event Codes

Event Code	Meaning	Event Code	Meaning
0	Null Event	16	In desk
1	Mouse down	17	In menu bar
2	Mouse up	18	In system windows
3	Key down	19	In content region
4	*Undefined*	20	In drag region
5	Auto key	21	In grow box
6	Update	22	In go away box
7	*Undefined*	23	In zoom box
8	Activate	24	In information bar
9	Switch	25	*Undefined*
10	Desk accessory	26	*Undefined*
11	Device driver	27	In window frame
12	Application #1	28	In special menu item (edit menu)
13	Application #2	29	*Undefined*
14	Application #3	30	*Undefined*
15	Application #4	31	*Undefined*

If a program implements a particular event type, a subroutine label should be defined using the EVENTDEF statement for that event. When TASKPOLL detects an event, the event code is used as an index into the event dispatch table to determine the subroutine which handles the event. If a subroutine label is defined, program control transfers to that subroutine. The subroutine should end with the

RETURN 0 statement. This special form of the RETURN statement is necessary to support the special calling mechanism for event-handling subroutines.

Event number 17 in the event dispatch table is a special case. If no event handling subroutine is defined for event 17 (In Menubar), TML BASIC assumes that the program has defined menu handling routines using the MENUDEF statement.

The second 32 entries in the event dispatch table (numbered 32 through 63) are used with the EXEVENT statement for obtaining the machine addresses of subroutines for implementing *definition procedures*. See the EXEVENT statement for more information regarding this use of the event dispatch table.

See Chapter 13 for a complete discussion on how to write event-driven, desktop applications.

See Also

EXEVENT
MENUDEF
TASKPOLL
Chapter 13

Example

```
EVENTDEF 8,doActivate
EVENTDEF 22,doCloseBox
```

EXCEPTION Statements

Syntax

```
EXCEPTION ON Mask
EXCEPTION OFF
EXCEPTION 0
```

Action

TML BASIC implements floating-point arithmetic operations using the SANE (Standard Apple Numeric Environment) mathematical routines and provides the programmer with control over the exceptions generated by the tool set. There are three modes available in handling these exceptions that can be selected by the EXCEPTION statement.

The default mode is selected with EXCEPTION OFF. Selecting any other option for the EXCEPTION statement should only be done if you have a complete understanding of what SANE exceptions are and how they work. In the default mode, TML BASIC returns the standard error messages for the important mathematical calculation exceptions and ignores the unimportant exceptions.

EXCEPTION 0 (zero) is used to disable all SANE exceptions and will cause all exceptions to be ignored and pass through as NaN's for expression results.

EXCEPTION ON is used to enable exception trapping in your program of a specific type beyond the normal default settings. The *SaneMask* parameter must be a number between 0 and 63 and is used as a mask to filter the SANE exceptions. The SANE halt vector is always enabled, and all halts are received by TML BASIC. The mask is used to determine if any specific exception will generate a BASIC error message or be ignored.

See Also

ON EXCEPTION

EXEVENT@ Function

Syntax

EXEVENT@(*EventCode*)

Action

The EXEVENT@ function returns the machine address for one of 32 external event entry points in the *Event Dispatch Table.* As described with the EXEVENT statement, the event dispatch table is a special data structure defined for directing program control to event-handling subroutines when an event occurs in a desktop application. The event dispatch table has 64 entries numbered 0 through 63. The first 32 entries (0 through 31) are reserved for use with the TASKPOLL statement. The remaining 32 entries are used with EXEVENT@.

The implementation of several Toolbox features requires the ability to directly call subroutines written in a TML BASIC program. The Toolbox defines these subroutines, *definition procedures*. For example, when creating a window with the NewWindow function, a program can specify a *Content definition procedure* for the window. The content definition procedure is automatically called by the Toolbox whenever the contents of the window need to be drawn or re-drawn.

When providing a definition procedure to the Toolbox, a machine address is required. To obtain the address of a subroutine label, the label is first entered into an element of the event dispatch table using the EVENTDEF statement. Then the EXEVENT@ function is used to obtain the address.

See Chapter 13 for a complete discussion on how to write event-driven, desktop applications.

See Also

TASKPOLL
EVENTDEF

Example

```
EVENTDEF 63, DrawMyWindowContent
defproc@ = EXEVENT@(63)
SET(WindowParamBlock!(58),4) = defproc@
```

EXFN_ Function

Syntax

```
    EXFN[%|@|&|#|$]_ToolName [( Expression {, Expression } ) ]
```

Action

The EXFN function executes a named procedure or function in an Apple IIGS toolset and returns a value. The declarations for Toolbox procedures and functions are defined in the several predefined libraries shipped with TML BASIC in the folder LIBRARIES. See Appendix C for a complete list of the Toolbox libraries and the procedures and functions declared in them.

Following the reserved word EXFN and the underscore character (_) is the name of the toolbox procedure or function to execute. If the routine has parameters, they are given after the toolbox name enclosed in parenthesis. The rules for matching parameters are the same as for normal BASIC procedures. If the Toolbox routine is a function, the EXFN function returns the result value. The result values are placed in the *CALL return stack*. If the Toolbox routine is a procedure, the EXFN function returns the Toolbox error code, indicating the success or failure of the operation. See the description of the reserved variable R.STACK for a description of the Call return stack.

In order to call a Toolbox procedure or function, the library containing the declaration of the routine must appear in a LIBRARY statement, otherwise TML BASIC reports the error "Toolbox procedure *xxx* is not defined", where *xxx* is the name of the procedure.

Chapter 11 provides a detailed discussion of the Apple IIGS Toolbox and how to access it from TML BASIC.

See Also

CALL
R.STACK
Chapter 11
Appendix C

Example

```
LIBRARY "Memory"                        'Load the Memory Manager library
MyID% = EXFN_MMStartUp                  'Start the memory manager
MyHndl@ = EXFN_NewHandle(1024,MyID%,0,0)  'Allocate a 1K block of memory
```

EXP, EXP1 and EXP2 Functions

Syntax

```
    EXP(x)
    EXP1(x)
    EXP2(x)
```

Action

The EXP function returns *e* to the *x* power, where *x* is a numeric expression and *e* is the base for natural logarithms (approximately equal to 2.718282). To calculate the exact value of *e* use EXP(1).

The EXP1 function accurately computes e^x-1. If the value of *x* is small, then the computation of EXP1 is more accurate than EXP(x)-1.

Finally, the EXP2 function returns 2 to the *x* power.

In all three functions, *x* is a numeric expression.

Example

```
FOR i% = 1 to 10
  PRINT i%, EXP(i%), EXP1(i%), EXP2(i%)
NEXT i%
```

FILE Function

Syntax

```
    FILE(PathName [, FILTYP=TXT|SRC|BDF|FileType ] )
```

Action

The FILE function is used to determine whether or not a file exists.

FILE is an integer function which returns the value one (1) if the file specified by the *PathName* string expression exists, otherwise it returns zero (0). If the optional FILTYP= parameter exists, the FILE function also checks that the file also has the filetype specified by the FILTYP= parameter. If the file exists, but the filetype does not match, then FILE returns zero (0).

If the file does exist then the AUXID@ reserved variable is updated to contain the subtype of the specified file, and the FILTYP(0) function call returns the file type of the specified file.

If an illegal ProDOS 16 pathname is specified in *PathName* then the "Bad Path Error" occurs. For a thorough description of the FILTYP= parameter see the CREATE statement.

See Also

AUXID@
CREATE
FILTYP
Chapter 9

Example

```
AFile$ = "AnyFile"

IF FILE(AFile$) THEN
   OPEN AFile$, AS #1
ELSE
   PRINT "The file ";AFile$;" does not exist and can not be open"
END IF
```

FILETYP Function

Syntax

```
    FILTYP(FileNumber)
```

Action

The FILTYP function returns the file type of a file previously opened with *FileNumber* as its file reference number. *FileNumber* is a numeric expression that must be an integer from 0 to 31, otherwise an "Illegal Quantity Error" occurs. If *FileNumber* is a legal file number, but no open files have the specified file number, the "File Not Open" error occurs.

FILTYP(0) is a special case that returns the file type of the last FILE function call.

See Also

FILE
Chapter 9

Example

```
OPEN "SOMEFILE", AS #5
PRINT "The file type for file #5 is "; FILTYP(5)
```

FIX Function

Syntax

```
    FIX(NumericExpression)
```

Action

The FIX function truncates the absolute value of *NumericExpression* and returns the signed integer portion. Note that this is different than the INT function which returns the next lower number for a negative *NumericExpression*.

FIX is equivalent to the expression: SGN(x) * (INT(ABS(x))

See Also

INT

Example

```
PRINT FIX(1.5), FIX(-1.5)
PRINT INT(1.5), INT(-1.5)
```

OUTPUT:

```
1          -1
1          -2
```

FN = Statement

Syntax

```
FN VariableName = AnyExpression
```

Action

The FN assignment statement is a special case of the assignment statement which can only be used within a multiline procedure or function. The purpose of the FN assignment statement is to ensure that the destination of the assignment statement is a local variable, a formal parameter or a function result variable, otherwise TML BASIC will give the "Not Local" error.

The purpose for this variation of the assignment statement is to ensure that an assignment statement within a multiline procedure or function does not reference a global variable. Even more important is to ensure that a *new* global variable is not created by an assignment. A side effect of this feature is that statements become *self-documenting* with respect to whether a local or global variable reference is being made.

See Also

DEF FN
DEF PROC
LOCAL
LET

Example

```
DEF FN Add% (Num1%, Num2%)
   LOCAL Temp%
   FN Temp% = Num1% + Num%
   FN Add% = Temp%
END FN Add%
```

FOR ... NEXT Statement

Syntax

```
FOR Counter = Start TO End [STEP Increment ]
   statements
NEXT [Counter {,Counter } ]
```

Action

The FOR...NEXT statement is a looping construct. The statement groups one or more statements and executes them repetitively, a specified number of times. *Counter* must be a numeric variable (not a string variable or array element) which is the loop control variable. *Start, End* and *Increment* must all be numeric expressions whose values are compatible with the type of the variable *Counter,* otherwise a "Type Mismatch Error" occurs.

When the FOR statement is first encountered, the value of *Start* is assigned to the variable *Counter,* and the values of *End* and *Increment* are evaluated and stored in a temporary location. If the optional STEP *Increment* does not appear, then a default *Increment* of one (1) is used. Following this, the sequence of statements after the FOR statement are executed until the NEXT statement is encountered. If the NEXT statement does not specify a *Counter* variable then it matches to the most recent FOR statement. If a *Counter* variable is given then it must match the *Counter* variable in the most recent unmatched FOR statement (FOR...NEXT statements may be nested).

The NEXT statement increments the *Counter* variable by the value of *Increment* and then tests to see if the loop should be repeated. If the value of *Increment* is positive then the NEXT statement checks to see if *Counter* is less than or equal to *End.* If the value of *Increment* is negative then the NEXT statement checks to see if *Counter* is greater than or equal to *End.* If this test passes, control loops back to the first statement after the matching FOR statement. This process continues until the test fails, execution then continues with the statement after the NEXT statement.

If a FOR statement does not have a matching NEXT statement then a "FOR Without Matching NEXT" error occurs. And likewise, if a NEXT statement does not have a matching FOR statement then a "NEXT Without Matching FOR" error occurs.

Example

```
FOR SKIP% = 0 TO 10 STEP 2        'Print even numbers between 0 and 10
  PRINT SKIP%
NEXT

FOR CountDown = 10 TO 1 STEP -1 'A simple countdown loop
   PRINT CountDown ; " " ;
NEXT
PRINT "DONE!"

FOR Row% = 1 TO 3                 'Nested FOR...NEXT loops
   FOR Column% = 1 TO 4
      PRINT "(" ; Row% ; "," ; Column% ; ")" ;
   NEXT Column%
   PRINT
NEXT Row%
```

OUTPUT:

```
0
2
4
6
8
10

10 9 8 7 6 5 4 3 2 1 DONE!

(1,1)(1,2)(1,3)(1,4)
(2,1)(2,2)(2,3)(2,4)
(3,1)(3,2)(3,3)(3,4)
```

FRE Reserved Variable

Syntax

```
    FRE
```

Action

FRE is a reserved variable that returns the amount of free memory available in the Apple IIGS.

Compiler/Interpreter Differences

In GS BASIC, the FRE reserved variable returns the amount of memory in the program data segment. Since TML BASIC can use all available memory in the Apple IIGS for data storage, the FRE reserved variable is redefined to return the amount of free memory in the machine rather than a special data segment.

See the description of the *$DSeg* metastatement in Appendix B for a description of data segmentation in TML BASIC.

See Also

FREMEM
Chapter 7

Example

```
'Show available memory
PRINT "Free memory in Apple IIGS: "; FRE

'Allocate a few large dynamic arrays
DIM DYNAMIC BigArray1%(1000), BigArray2@(400)

'Now print available memory...
PRINT "Free memory in Apple IIGS: "; FRE
```

FREMEM Function

Syntax

```
FREMEM(NumericExpression)
```

Action

The FREMEM function is used to return information about the use of memory in the Apple IIGS. The *NumericExpression* parameter must be an integer value in the range 0 through 9, otherwise an "Illegal Quantity Error" occurs. The information returned by FREMEM depends upon the value of *NumericExpression*.

The meaning of FREMEM for each value of *NumericExpression* is defined as follows:

0-6	Returns the total available free memory in the Apple IIGS. (Same as FRE)
7	Returns the total available free memory in the Apple IIGS after performing a Memory Manager CompactMem call.
8	Returns the size of the Memory Manager's largest free contiguous block of memory.
9	Returns the total memory installed in the Apple IIGS.

Compiler/Interpreter Differences

In GS BASIC, the FREMEM function returns special values related to the way the GS BASIC interpreter executes programs when the value of *NumericExpression* is in the range 0 through 6. The values returned indicate such things as the size of the program, the size of the data segment, the size of the library segment, and other data structures not implemented by TML BASIC. Because TML BASIC is a compiler which produces stand-alone programs, the various data structures implemented by GS BASIC to execute a program are not required. In these cases, the value of FREMEM is the same as FRE.

GET# Statement

Syntax

```
    GET# FileNumber [,[Length] [,RecordNumber] ]; StructureVariable
```

Action

The GET# statement reads a record from the binary file previously opened with *FileNumber* as its file reference number and stores the data into the *StructureVariable*. The *StructureVariable* may have an index expression. The GET# statement can read data from a file of any file type.

The number of bytes read by the GET# statement is determined by the *RecordSize* parameter specified in the OPEN statement. If the *RecordSize* was not specified in the OPEN statement then the file's subtype is the record size. The number of bytes read may be overridden by specifying the optional *Length* parameter in the GET# statement. You should not attempt to read more data than the *StructureVariable* can hold.

The GET# begins reading at the current position in the file. To begin at a random record position, the optional *RecordNumber* parameter must be used.

See Chapter 9 for a complete discussion of Files in TML BASIC.

See Also

OPEN
PUT#
Chapter 9

Example

```
DIM  myData!(11)
OPEN "SOMEFILE", FILTYP=0 AS #1, 4 'Open a binary file whose record size is 4

GET #1; myData!(0)                 'Read first 4 bytes of file
GET #1,,3; myData!(4)              'Read 4 bytes starting at record 3
GET #1,12,5; myData!(0)            'Read 12 bytes starting at record 5

CLOSE #1
```

GET$ Statement

Syntax

```
    GET$ [# FileNumber [,RecordNumber]];StringVariable
```

Action

The GET$ statement reads a single character into the *StringVariable*.

By default, GET$ reads the character from the keyboard, without displaying it to the screen, and without waiting for the Return key to be pressed.

If the optional *FileNumber* appears, then the GET$ statement reads the character (a single byte) from the file previously opened with *FileNumber* as its file reference number. The *RecordNumber* option allows the character to be read from the beginning of specified record.

Since files can contain values that are not defined as ASCII characters, it is the responsibility of the program to ensure the file contains valid characters. For example, reading a byte with a value of zero may cause unpredictable results later when using the string.

GET$ treats Control-C like any other character; it does not interrupt program execution.

Example

```
PRINT "Press any key to continue.";
GET$ A$
```

GOSUB Statement

Syntax

`GOSUB` *`Label`*

Action

The GOSUB statement causes execution to temporarily suspend and branch to the statement indicated by *Label*. When the subsequent sequence of statements encounters a RETURN statement, execution branches back to the statement immediately after the most recently executed GOSUB.

The group of statements indicated by the *label* and the RETURN statement are collectively called a subroutine. Subroutines provide BASIC programmers an effective means to organize their code into logically organized components. A subroutine may call another subroutine, which in turn may call yet other subroutines. TML BASIC automatically keeps track of where execution should resume when the RETURN statement is executed.

See Also

ON...GOSUB
POP
RETURN
Chapter 7, Labels
Chapter 8, Subroutines

Example

```
GOSUB Sub1
END
Sub1: PRINT "In Subroutine 1"
      GOSUB Sub2
      PRINT "Leaving Subroutine 1"
      RETURN

Sub2: PRINT "In Subroutine 2"
      PRINT "Leaving Subroutine 2"
      RETURN
```

OUTPUT:

```
In Subroutine 1
In Subroutine 2
Leaving Subroutine 2
Leaving Subroutine 1
```

GOTO Statement

Syntax

```
    GOTO Label
```

Action

The GOTO statement causes execution to unconditionally branch to the statement indicated by *Label*. It is normally considered better programming practice to use TML BASIC's structured control statements such as the DO...WHILE...UNTIL, IF...THEN and FOR...NEXT statements rather than the GOTO statement. GOTO statements generally make programs difficult to read and debug.

See Also

ON...GOTO
Chapter 7, Labels

Example

```
GOTO CalculateAverage
GOTO TryAgain
```

GRAF INIT, GRAF OFF and GRAF ON Statements

Syntax

```
    GRAF  INIT 0 | 320 | 640
    GRAF  OFF
    GRAF  ON
```

Action

The GRAF statements are used to initialize, turn on and turn off the Apple IIGS Super Hi-Res graphics screen.

The GRAF INIT statement must be called before GRAF ON or GRAF OFF, and before any calls to the QuickDraw routines using CALL or CALL%. The GRAF INIT statement allocates the memory needed for the QuickDraw graphics engine and properly initializes it. If the value following GRAF INIT is 320 then the Super Hi-Res screen is placed in 320 mode. If the value is 640 then the Super Hi-Res screen is placed in 640 mode. If a value of 0 is specified then QuickDraw is shut down, the Super Hi-Res screen is turned off and the text screen becomes active. Note that the GRAF INIT statement does not initialize the QuickDraw Auxiliary tool set. If a program uses any of these routines, it is responsible for loading and properly initializing QuickDraw Auxiliary tool set.

The GRAF ON statement is used to make the Super Hi-Res screen the current screen mode. GRAF INIT 320 or GRAF INIT 640 must have already been called. This statement is the same as the QuickDraw _GrafOn procedure.

The GRAF OFF statement is used to temporarily turn off the Super Hi-Res screen and make the text screen the current screen mode. Again, GRAF INIT 320 or GRAF INIT 640 must have already been called. This statement is the same as the QuickDraw _GrafOn procedure.

Example

```
GRAF INIT 640     'Initialize QuickDraw with the 640 Super Hi-Res screen
GRAF ON           'Turn on the Super Hi-Res graphics screen
_ClearScreen(-1)  'Make the screen white
_LineTo(60,45)    'Draw a line
GRAF OFF          'Turn the Super Hi-Res graphics screen off
GRAF INIT 0       'Shutdown QuickDraw
```

HEX$ Function

Syntax

```
    HEX$(NumericExpression)
```

Action

The HEX$ function returns an eight character string which is the hexidecimal (base 16) representation of *NumericExpression*. If the hexadecimal representation requires fewer than 8 digits then leading zeros are inserted so that 8 characters are always returned. *NumericExpression* must be in the range -2^32 to 2^32-1 or else an "Illegal Quantity Error" occurs.

Example

```
PRINT HEX$(32767)
PRINT HEX$(10)
```

OUTPUT:

```
00007FFFF
00000000A
```

HOME Statement

Syntax

```
    HOME
```

Action

Clears the contents of the current text window and places the cursor in the upper left corner of the text window. Note that HOME only clears the contents of the current text *window*. By default the text window is the entire text screen, however, this can be changed using the TEXTPORT statement.

See Also

HPOS and VPOS
TEXTPORT

Example

```
HOME
```

HPOS and VPOS Modifiable Reserved Variables

Syntax

```
    HPOS
    HPOS = NumericExpression

    VPOS
    VPOS = NumericExpression
```

Action

HPOS and VPOS are modifiable reserved variables which contain the horizontal and vertical positions, respectively, of the current text screen cursor position. In addition, the variables may be assigned new values to change the current cursor position.

Assigning a value greater than the height of the current text window causes the cursor to move to the bottom line within the text window. Likewise, assigning a value greater than the width of the current text window causes the cursor to move to the right margin of the text window. In any case, the value of *NumericExpression* must be within 0 to 255 inclusive or an "Illegal Quantity Error" occurs. Note that the text window is normally the entire text screen, however, this can be changed using the TEXTPORT statement.

See Also

TEXTPORT

Example

```
HPOS = 10
VPOS = 21

PRINT "The current cursor position is (" ; HPOS ; "," ; VPOS ; ")"
```

OUTPUT:

```
The current cursor position is (10,21)
```

IF...THEN...ELSE Statement
IF...GOTO Statement

Syntax

```
    IF Expression THEN StatementList [:ELSE StatementList ]

    IF Expression GOTO Label
```

Action

The IF statement in TML BASIC forms a structure for deciding which statements in a program to execute. An IF statement has a condition (any legal expression) which may contain relational operators like < and > (less than and greater than), logical operators like OR and AND, and arithmetic operators. If the condition is true (any non-zero value), TML BASIC executes the statements following the THEN. If the condition is false (a zero value), TML BASIC ignores the statements following the THEN.

The simplest form of the IF statement is the single-line IF statement. For example:

```
    IF RND(1) < 0.5 THEN PRINT "Heads, you win"
```

In this statement, the expression *RND(1) < 0.5* is evaluated. If the expression is true, the statements following the reserved word THEN are executed; otherwise control passes to the statement after the IF statement.

TML BASIC provides several other variations of the IF statement. The IF...THEN...ELSE statement is the simplest of these variations. The ELSE part of the statement allows a program to specify statements to be executed only when the IF condition is false. For example,

```
    IF RND(1) < 0.5 THEN PRINT "Heads, you win" :ELSE PRINT "Tails, I win"
```

Notice that a colon must precede the reserved word ELSE.

TML BASIC allows this statement to be rewritten on two lines as follows:

```
    IF RND(1) < 0.5 THEN PRINT "Heads, you win"
      ELSE PRINT "Tails, I win"
```

In addition, the IF statement can be rewritten on three lines as follows:

```
    IF RND(1) < 0.5
      THEN PRINT "Heads, you win"
      ELSE PRINT "Tails, I win"
```

In each of these last two formats, the THEN and ELSE statements must be on the lines immediately following the IF statement, and the statements following the reserved words THEN and ELSE must fit on one line.

The final variation of the IF statement is the IF...GOTO statement. When an IF statement has only a THEN part, and the only statement following the reserved word THEN is a GOTO statement, the IF...GOTO statement can be used. For example,

```
IF RND(1) < 0.5 GOTO PrintMsg
```

See Also

IF Block

IF Block Statement

Syntax

```
IF Expression THEN
   •
   • Statements
   •
{ ELSEIF Expression THEN
   •
   • Statements
   • }
[ ELSE
   •
   • Statements
   • ]
END IF
```

Action

A much more powerful variation of the IF statement available in TML BASIC is the *IF Block* statement. This variation of the IF statement allows programs to place the statements normally appearing after the reserved word THEN on one or more lines *after* the IF statement. An IF block is ended by the END IF statement. The following example illustrates how the IF block statement can be used.

```
IF RND(1) < 0.5 THEN
   PRINT "Heads, you win"
   CountHeads = CountHeads + 1
END IF
```

If the expression *RND(1) < 0.5* is true then all the statements between the IF and the END IF are executed. If the expression is false, control passes to the statement after the END IF statement.

The IF block statement can be used to create even more powerful control structures using the ELSE statement. In the following example, when the expression *RND(1) < 0.5* is false control passes to the statement after the ELSE statement. The ELSE statement also marks the end of the THEN part as well.

```
IF RND(1) < 0.5 THEN
   PRINT "Heads, you win"
   CountHeads = CountHeads + 1
ELSE
   PRINT "Tails, I win"
   CountTails = CountTails + 1
END IF
```

Finally, the IF block statement can be used with the ELSEIF statement to create sophisticated control structures. The ELSEIF statement allows the program to create multi-part IF statements, each having a different condition to satisfy. The following example illustrates how the tosses of a "three headed" coin might be recorded.

```
IF RND(1) < 0.3 THEN
   PRINT "Head 1, you win"
   CountHead1 = CountHead1 + 1
ELSEIF RND(1) < 0.6 THEN
   PRINT "Head 2, I win"
   CountHead2 = CountHead2 + 1
ELSE
   PRINT "Head 3, someone else wins"
   CountHead3 = CountHead3 + 1
END IF
```

This variation of the IF statement allows complex branching be added to a program. If the first condition contained in an IF block statement is not true, control is immediately passed to the next ELSE or ELSEIF statement until either a true condition is met or the ENDIF statement is encountered.

See Also

IF...THEN...ELSE

IMAGE Statement

Syntax

```
    IMAGE Specification {, Specification }
```

Action

The IMAGE statement is used to control formatting of print items in the PRINT USING and PRINT# USING statements. In the following paragraphs the statement PRINT USING implies both PRINT USING and PRINT# USING.

The PRINT USING statements include a *Using Specification* which is used by TML BASIC to control the formatting of print items in the statement. A Using Specification contains one or more individual *Specifications,* each corresponding to an individual print item. The Using Specification can be included directly in the PRINT USING statement in the form of a string constant, string variable or as a label reference to an IMAGE statement. Regardless of the way a *Specification* is defined, the formatting of information is the same. For example, the following forms of a Using Specification are equivalent:

```
PRINT USING "5C, ###.##"; Msg$, Number

PrintSpec$ = "5C, ###.##"
PRINT USING PrintSpec$; Msg$, Number

PRINT USING PrintImage; Msg$, Number
PrintImage: IMAGE 5C, ###.##
```

A *Specification* is a collection of special letters, numbers and/or symbols which define a formatting code. Each individual specification must be separated by a comma. However, note that the commas in the PRINT USING statement only serve to separate the individual print items, they do not cause a tab action to the next print zone as in the PRINT statement.

There are three different types of PRINT USING *Specifications:* string specification, literal specification and numeric specification. A string specification controls the formatting of string values in a PRINT USING statement. A literal specification inserts either one or more spaces, one or more line returns or one or more specified characters into the text displayed by the PRINT USING statement. Finally, a numeric specification controls the formatting of numeric values in a PRINT USING statement. The following paragraphs describe each of these different types of specifications.

A String Specification defines the field format and width for a string value. Three formats are available: left-aligned, centered, right-aligned. The codes for these formats are as follows:

A	Left-aligned
R	Right-aligned
C	Centered

The width of the string can be defined by either specifying the number of characters in the field, or by preceding the specification character with an integer which is the width value. For example, the following two specifications define a 6 character centered string value:

```
CCCCCC
6C
```

The integer preceding the specification character is called a *repeat factor*, and only affects the single character immediately following it. For example, the following specifications also define a 6 character centered string value.

```
4CCC
CCC3C
```

A repeat factor must be in the range 1 through 255. If a string value exceeds the string specification, its value is truncated.

A Literal Specification does not format any value contained in the PRINT USING statement, but rather inserts one or more spaces, line returns or specified characters into the printed text. There are three literal specification codes:

X	Prints a space
/	Prints a line return
" "	Encloses a literal string

Again, a repeat factor can be used with the codes. For example,

4X	Prints 4 spaces
2/	Prints 2 line returns
4"ab"	Prints: abababab

Finally, a Numeric Specification formats numeric values in fixed-point, scientific or engineering formats. There are three numeric specification codes which are used in all three numeric formats. They are called the digit specification codes.

#	Reserves one numeric digit position, suppresses leading zeros
Z	Reserves one numeric digit position, prints leading zeros
&	Reserves one position for a digit or a comma

Again, a repeat factor can be used with the specification characters.

The fixed-point numeric specification controls the formatting of fixed-point numbers. Fixed-point numbers are any numbers displayed without exponents, both integer and real. TML BASIC provides additional specification characters for fixed-point numbers.

+	Reserves one character position for a number sign (+ or -)
-	Reserves one character position for a minus sign if negative
$	Reserves one character position for a dollar sign
**	Prints asterisks instead of leading spaces
++	Reserves rightmost positions for a dollar sign (if any)
--	Same as ++, except minus sign printed only if number is negative
$$	Reserves leftmost positions for a dollar sign (if any)

If the ** specification characters are used, they must be the first characters, and should only be used with # and & since the Z specification character leaves no unused digit positions. If the width of a numeric specification is insufficient for the number of digits required to display a value, the width of the display is filled with exclamation points (!).

To format numeric values in scientific notation, the E specification character is used to define the width of the exponent. Scientific notation contains only one or zero digits to the left of the decimal point, then the desired number of significant digits to the right of the decimal point, followed by the number of digits for the exponent. The width of the exponent must be at least three or four character positions. The following are legal scientific notation specifications:

```
#.#####EEEE
#.5#4E
.6#3E
+.#########4E
```

Engineering notation specifications are defined using a variation of the scientific notation specifications. In scientific notation specifications, only one or zero digit positions are permitted to the left of the decimal point. Engineering specifications, on the other hand, specify three digit positions. However, the number of digits actually displayed varies so that the exponent value is always a multiple of 3. For example, the following are legal engineering notation specifications because the number of digit positions to the left of the decimal point is three.

```
3#.4#4E
###.####EEEE
###.2#4E
```

Thus, only zero, one or three digit positions are permitted in a numeric specification that includes an exponent. If the number of digits is zero or one, scientific notation is used, if the number of digits is three, engineering notation is used.

If a Using Specification contains an illegal specification (ie. illegal code characters, or improper use of legal characters), the runtime error "Illegal Using Specification" occurs.

See Also

PRINT USING
PRINT# USING

Example

```
PRINT USING BigImage; "BASIC", "BASIC", "BASIC"
BigImage: "123456789", 9A, /, 9R, /, 9C     'Literal and String specifications

PRINT USING "####, /"; 1,22,333,4444        'Fixed point specifications
PRINT USING "$####.##, /"; 23.4, 1293.32

PRINT USING "3#.4#4E"; 123456               'Engineering specifications
PRINT USING "3#.4#4E"; 1234567

PRINT USING "3Z.4Z4E"; 123456               'Engineering specifications
PRINT USING "3Z.4Z4E"; 1234567
```

OUTPUT:

```
123456789
BASIC
    BASIC
  BASIC

   1
  22
 333
4444

$  23.40
$1293.22

123.4560E+03
  1.2345E+06

123.4560E+03
001.2345E+06
```

INPUT Statement

Syntax

```
INPUT [StringConstant ,|;] VariableName {,VariableName }
```

Action

The INPUT statement is used to obtain one or more numeric or text values entered at the keyboard. When the INPUT statement is executed, TML BASIC accepts one or more values entered from the keyboard and assigns them into the variables listed in the INPUT statement. When more than one variable is listed in an INPUT statement, each of the values entered at the keyboard must be separated by a comma or a Return key.

When the INPUT statement is executed, a question mark (?) is displayed on the screen indicating the program is waiting for input. If a Return key is entered and variables still exist which have not been given values, TML BASIC displays two question marks (??) indicating more data is required by the INPUT statement.

The INPUT statement may also contain a string which is displayed as the input prompt instead of the normal question mark. The string must appear immediately after the reserved word INPUT and must be a string constant and *not* a string variable or expression.

The INPUT statement also works in the Super Hi-Res graphics screen. When the INPUT statement is executed, it examines the current screen mode. If the screen is in text mode (the default), text is input in the normal fashion. However, if the screen is in graphics mode, text is input from the current *GrafPort* (window) using QuickDraw graphics calls. Text is drawn beginning at the current location of the QuickDraw pen. None of the TML BASIC screen position commands work in the graphics screen. To move the pen, QuickDraw commands such as Move and MoveTo must be used. For more information about QuickDraw see Chapter 12.

See Also

INPUT USING
PRINT

Example

```
REM A program to compute the average of three numbers
INPUT "Enter three numbers: "; Number1, Number2, Number3
Avg = (Number1 + Number2 + Number3) / 3
PRINT "The average of the three numbers is "; Avg
GET$ Key$
```

INPUT# Statement

Syntax

`INPUT# FileNumber [,RecordNumber] [;VariableName {,VariableName }]`

Action

The INPUT# statement reads a *line* of text from a file into an input buffer and then processes the input text according to the list of input variables in its argument list. If the INPUT# statement does not encounter a return character after reading 255 characters, it terminates reading the file, appends a return character to the input buffer, and processes the characters as a single line.

FileNumber is a file reference number of an open text file. The list of comma separated *VariableNames* may be both string and numeric variables. If a numeric variable is used in an INPUT# statement, TML BASIC automatically converts the string representation of a number into the appropriate numeric type (similar to the VAL statement). When a numeric variable is used in an INPUT# statement and the input line does not contain a string which represents a legal numeric value a "Type Mismatch Error" occurs. If there is not enough data in the input line, the file is read again until all of the variables have been given values.

If the optional *RecordNumber* argument does not appear, the INPUT# statement reads sequentially beginning at the current file position. To perform random access using the INPUT# statement, include a record number after the file reference number. Recall that the file must be opened using the OPEN statement with the optional record size argument specified in order to define the size of a record for the text file.

See Also

PRINT
Chapter 9

Example

```
DEF PROC ReadFile(FileNam$)
   LOCAL aLine$
   OPEN FileNam$, AS #1
   ON EOF #1 GOTO Finished
   NextLine: INPUT #1; aLine$
             PRINT aLine$
             GOTO NextLine
   Finished: CLOSE #1
END PROC
```

INSTR Function

Syntax

```
INSTR(String1, String2 [,NumericExpression])
```

Action

The INSTR function searches for the first occurrence of the substring designated by the string expression *String2* in the string *String1* and returns the starting position of the substring. If the substring *String2* does not exist in the string *String1* the search fails and returns the value zero (0). Note that the search is case sensitive.

If the optional *NumericExpression* is present, it specifies the character position within *String1* where the search should begin. If a *NumericExpression* is not present the search begins at the first character of *String1*. If the value of *NumericExpression* is less than 1 or greater than the length of the string then an "Illegal Quantity Error" occurs.

Example

```
PRINT INSTR("TML BASIC is great", "basic")
PRINT INSTR("TML BASIC is great", "BASIC")
PRINT INSTR("TML BASIC is great", "BASIC", 10)
```

OUTPUT:

```
0
5
0
```

INT Function

Syntax

```
    INT(NumericExpression)
```

Action

Returns the largest whole number less than or equal to the value of *NumericExpression*. The whole number returned is actually a real value and not an integer. This function is often misunderstood for negative numbers, see example.

See Also

FIX

Example

```
PRINT FIX(1.5), FIX(-1.5)
PRINT INT(1.5), INT(-1.5)
```

OUTPUT:

```
1          -1
1          -2
```

INVERSE Statement

Syntax

```
    INVERSE
```

Action

The INVERSE statement is used to change the display of all subsequent characters written to the text screen using "inverse video". If you are using a monochrome display, INVERSE causes characters to be displayed as black on a white background. If you are using a color display then the effect depends upon the settings of your monitor. In this case, it is more appropriate to use the terms text background and foreground color.

INVERSE does not affect any characters which are already displayed on the screen, only the screen output after INVERSE is executed. INVERSE does not effect characters written to files.

See Also

NORMAL

Example

```
NORMAL
PRINT "This is normal display"
INVERSE
PRINT "This is inverse display"
```

JOYX Function
JOYY Reserved Variable

Syntax

> JOYX(*PaddleNumber*)
> JOYY

Action

The JOYX and JOYY functions are used to read the current value of the game paddles.

JOYX reads two of the four game paddle inputs (if they are plugged in) specified by *PaddleNumber*. *PaddleNumber* must be an integer in the range 0 to 2, otherwise an "Illegal Quantity Error" occurs. JOYX reads the value of the indicated paddle, returns the value, and also sets the JOYY reserved variable. The reserved variable JOYY is set to the value of the paddle indicated by *PaddleNumber* + 1.

Example

```
       HOME
Start: PRINT JOYX(1), JOYY
       GOTO Start
```

OUTPUT:

```
12          34            these are paddle values as paddle knobs are turned
12          55
12          34
12          55
12          34
12          55
12          34
12          55
12          34
12          55
12          34
12          55
```

KBD Reserved Variable

Syntax

```
KBD
```

Action

The KBD reserved variable contains an integer value which is the ASCII code of the last key pressed from the keyboard. A table of the ASCII codes can be found in Appendix E.

When using the KBD reserved variable in the ON...GOTO or ON...GOSUB statement, it must be enclosed in parenthesis in order to create an expression syntax and thus distinguish between these statements and the ON KBD statement. For example, the following statement is treated as the ON KBD statement which turns on event trapping for keypresses.

ON KBD GOTO HandleKeyPress

However, the following is the ON...GOTO statement which branches to the label DoIt if the value of the reserved variable KBD is one (1).

ON (KBD) GOTO DoIt

See Also

ON KBD

Example

```
ON KBD GOTO ShowKey                     'Activate keyboard event trapping

Wait: GOTO Wait                         'Infinite loop to wait for keypresses

ShowKey: PRINT "The key = ";KBD
         IF KBD = ASC(".") THEN END     'Quit when the period is pressed
         ON KBD GOTO ShowKey            'Reactivate keyboard event trapping
         RETURN
```

LEFT$ Function

Syntax

```
    LEFT$(StringExpression, NumericExpression)
```

Action

The LEFT$ function returns the *NumericExpression* string of characters appearing left-most in the string *StringExpression.*

StringExpression may be any string variable, string constant or string expression. If *NumericExpression* is a real value, it is rounded to the nearest whole number. The value of *NumericExpression* must be an integer in the range 1 through 255 inclusive or an "Illegal Quantity Error" occurs. To find the number of characters in the string, use the LEN function.

See Also

LEN
MID$
RIGHT$

Example

```
PRINT LEFT$("TML BASIC is great",9)
```

OUTPUT:

```
TML BASIC
```

LEN Function

Syntax

```
    LEN(StringExpression)
```

Action

The LEN function returns an integer which is the number of characters in the *StringExpression*.

Example

```
Name$ = "TML BASIC"
PRINT LEN(Name$)
```

OUTPUT:

```
9
```

LET Statement

Syntax

```
[LET] VariableName = AnyExpression
```

Action

The LET statement, also known as the assignment statement, assigns the value of *AnyExpression* to the variable *VariableName*. *VariableName* may be any simple variable or array element. Only one assignment per statement is allowed. Note that the reserved word LET is optional.

If the type of the variable *VariableName* is a numeric type then *AnyExpression* may be any numeric type. TML BASIC automatically converts the value of *NumericExpression* to the type of the variable if the numeric types do not match. Finally, if the value cannot be represented in this type then an "Overflow Error" occurs. A "Type Mismatch Error" occurs if *AnyExpression* is a string expression.

If the variable is a string, then *AnyExpression* must also be a string, otherwise, a "Type Mismatch Error" occurs.

See Also

FN =
Chapter 7, Arrays

Example

```
LET Value1 = 30
LET Value2 = 23
LET Value3 = 8
LET Sum = Value1 + Value2 + Value3
```

OR

```
Value1 = 30
Value2 = 23
Value3 = 8
Sum = Value1 + Value2 + Value3
```

LIBRARY Statement

Syntax

```
    LIBRARY [PathName ]
```

Action

The LIBRARY statement is used in a program to load a compiled library file to memory and enter all of the procedure and function declarations of the library into the program's symbol table just as if the declarations had been made in the source code. There are two types of libraries in TML BASIC: user defined libraries created with the DEF LIBRARY statement, and predefined libraries which provide access to the Toolbox.

The LIBRARY statement can appear anywhere in a program. Before TML BASIC compiles a program, it first scans the file for all occurrences of the LIBRARY statement. As each LIBRARY statement is encountered, its declarations are entered into the program's symbol table, making them available throughout the entire program.

LIBRARY statements which name predefined Toolbox libraries serve a second purpose. As described in Chapter 11, several of the Toolbox tool sets are not available in ROM, but rather are implemented in disk files which must be loaded into RAM. When a LIBRARY statement names a tool set which is not in ROM, TML BASIC automatically generates code to load the disk file into RAM.

When a library name is specified in the LIBRARY statement, TML BASIC searches for the library's compiled *library file.* The library file is not the source code for the library, but its compiled declarations and code. As described in Chapter 3, the name for a library file is the name of the library with the suffix ".LIB". For example, the library filename for the toolbox library QuickDraw is QUICKDRAW.LIB.

TML BASIC searches in three locations to find a library file. First, it looks to see if the library file is already in memory. Second, it searches in the same folder as the source code file containing the LIBRARY statement. And finally, if the file is not found there, it searches in the directory specified in the *Library Search Path* option of the Preferences Dialog (see Chapter 6 for more information about the Preferences Dialog). If the file is not found in any of these locations then TML BASIC reports an error. However, it is possible to override TML BASIC by specifying the complete pathname of the library file.

See Also

DEF LIBRARY
Chapter 8
Chapter 11

Example

```
LIBRARY "Memory"
LIBRARY "QuickDraw"
LIBRARY "/TML/LIBRARIES/QUICKDRAW"
```

LOCAL Statement

Syntax

```
LOCAL VariableName {,VariableName }
```

Action

The LOCAL statement is only allowed within a multiline procedure or function. LOCAL is used to declare a simple variable as a temporary variable local only to the procedure or function. Local arrays are not supported.

When a procedure or function is called, the storage for the local variables is temporarily allocated and initialized to zero or the null string. When the procedure or function exits, the storage is deallocated. Local variables do not retain their values between calls. It is good programming practice to use the FN = variation of the assignment statement within a procedure or function to ensure that only local variable references are made. Using the FN = statement also promotes self-documenting code.

See Also

DEF FN
DEF PROC
FN =
Chapter 8, Local Variables

Compiler/Interpreter Differences

GS BASIC allows the LOCAL statement to appear anywhere among the statements of a multiline procedure or function. In fact, the LOCAL statement may even appear after an IF statement so that a local variable is conditionally declared.

TML BASIC restricts the use of the LOCAL statement. The LOCAL statements in a procedure or function must appear immediately after the DEF PROC or DEF FN statements and before any other statements with the exception of the REM statement.

Example

```
DEF PROC AverageThree(Val1,Val2,Val3)
LOCAL Sum
LOCAL Average
FN Sum = Val1 + Val2 + Val3
FN Average = Sum / 3
PRINT "The average is "; Average
END PROC AverageThree
```

LOCATE Statement

Syntax

```
    LOCATE [Row][,Column]
```

Action

The LOCATE statement is used to change the horizontal and vertical position of the text screen cursor. This statement essentially duplicates the functionality of the HPOS and VPOS reserved variables.

Both *Row* and *Column* must be numeric expressions. The *Row* argument changes the vertical position of the cursor to the specified value and should be in the range 1 through 24 inclusive. The *Column* argument changes the horizontal position of the cursor and should be in the range 1 through 80 inclusive. Both the *Row* and the *Column* arguments are optional. If only one of the arguments appears, the other component of the cursor position is unaffected. Of course, using the LOCATE statement without either of the arguments is meaningless, but legal.

See Also

HPOS and VPOS

Example

```
LOCATE 5,2  : PRINT "Hello"    'Change vertical and horizontal cursor position
LOCATE ,10  : PRINT "Goodbye"  'Change only the horizontal cursor position
LOCATE 6    : PRINT "Good Day" 'Change only the vertical cursor position
```

LOCK AND UNLOCK Statement

Syntax

```
LOCK PathName
UNLOCK PathName
```

Action

The LOCK and UNLOCK statements are used to change a file's write protection.

The LOCK statement prohibits writing to, saving, or deleting the file named in *PathName*. *PathName* is a string expression and must represent a legal ProDOS 16 pathname. A volume cannot be locked but subdirectories can. Any subsequent attempt to change the contents of a locked file will result in the "File Locked" error.

UNLOCK removes the protection placed upon a file by the LOCK statement. An unlocked file may be deleted, renamed, changed, or saved.

See Also

Chapter 9, Files

Example

```
LOCK myFile$
UNLOCK myFile$

LOCK "/TML/PART1.EXAMPLES/HELLOWORLD.BAS"
```

LOG, LOGB%, LOG1 and LOG2 Functions

Syntax

```
    LOG(x)
    LOGB%(x)
    LOG1(x)
    LOG2(x)
```

Action

The LOG function returns natural logarithm of *x*. The natural logarithm is to the base *e*.

The LOGB% function returns the binary exponent of the real value *x* as a signed integer.

The LOG1 function accurately computes the natural logarithm of *x+1*. If *x* is small, then the computation of LOG1 is more accurate than LOG(x+1).

Finally, the LOG2 function returns the base 2 logarithm of *x*.

In all four functions, *x* is a numeric expression.

Example

```
PRINT LOG(EXP(1))
PRINT LOGB%(100.0)
PRINT LOG1(EXP(1))
PRINT LOG2(32)
```

OUTPUT:

```
1
6
1.313262
5
```

MENUDEF Statement

Syntax

```
    MENUDEF ItemNumber, Label {,Label}
```

Action

The MENUDEF statement is used to store subroutine labels in the *Menu Item Dispatch Table.* The menu item dispatch table is a special data structure defined by TML BASIC for directing program control to menu item handling subroutines. The MENUDEF statement works in conjunction with the TASKPOLL and EVENTDEF statements.

The menu item dispatch table has 128 entries numbered 0 through 127. As discussed in Chapter 13, every *menu* contains one or more *menu items.* Each menu item has associated with it a unique *menu item identification number* (menu item id). The menu item ids for menus created by TML BASIC programs must be in the range 250 through 377 inclusive. These menu item ids correspond directly to the entries in the menu item dispatch table. The mapping of menu item ids to dispatch table entries is performed by subtracting 256 from the menu item id value.

When the TASKPOLL statement is executed and detects the *In Menu Bar* event, control is transferred to the menu item handling subroutine specified in the menu item dispatch table. For this to occur, the Event Dispatch Table for the In Menu Bar event must be zero (index 17), otherwise, control transfers to the event handling subroutine specified there. If index 17 of the event dispatch table is empty, TASKPOLL subtracts 250 from the menu item id of the selected menu item and looks up the menu item handling subroutine in the Menu Dispatch Table. If a subroutine has been defined, control transfers to the specified subroutine, otherwise TASKPOLL continues normal execution.

Menu item handling subroutines end with the RETURN 0 statement rather than the normal RETURN statement.

See Also

EVENTDEF
TASKPOLL

Example

```
MENUDEF 0,doNew
MENUDEF 1,doOpen
MENUDEF 2,doClose
```

MID$ Function

Syntax

```
    MID$(StringExpression, Start [,Length ] )
```

Action

The MID$ statement returns a string of *Length* characters from *StringExpression,* beginning with the *Start* character.

Start and *Length* must be numeric expressions whose values are in the range 1 through 255, otherwise an "Illegal Quantity Error" occurs. If the *Length* parameter does not appear, or if there are fewer characters to the right of the *Start* character then MID$ returns all of the right-most characters. If *Start* is greater than the number of characters in the string, MID$ returns a null string.

To determine the number of characters in a string use the LEN function.

See Also

LEFT$
LEN
RIGHT$

Example

```
INPUT "Binary number = "; Binary$       'Input a binary number as a string

DecimalVal@ = 0
FOR i% = 1 to LEN(Binary$)
   Digit$ = MID$(Binary$,i%,1)          'Get individual digit
   DecimalVal@ = 2 * DecimalVal@ + VAL(Digit$)
NEXT i%

PRINT "Decimal number = "; DecimalVal@
```

OUTPUT:

```
Binary number = 101001
Decimal number = 41
```

NEGATE Function

Syntax

```
NEGATE(NumericExpression)
```

Action

The NEGATE function returns the negation of *NumericExpression* (ie. - *NumericExpression)*. This seemingly simple function is included in TML BASIC because of the special infinity and NaN results possible using the Apple IIGS SANE floating point engine.

See Also

Chapter 7

Example

```
someValue = 5.2394
PRINT NEGATE(someValue)
```

OUTPUT:

```
-5.2394
```

NORMAL Statement

Syntax

```
    NORMAL
```

Action

The NORMAL statement is used to change the display of all subsequent characters written to the text screen in "normal video" (as opposed to inverse video). If you are using a monochrome display, NORMAL causes characters to be displayed as white on a black background. If you are using a color display then the effect depends upon the settings of your monitor. In this case, it is more appropriate to use the terms text background and foreground color.

NORMAL does not affect any characters which are already displayed on the screen, only the screen output after NORMAL is executed. NORMAL does not effect characters written to files.

See Also

INVERSE

Example

```
NORMAL
PRINT "This is normal display"
INVERSE
PRINT "This is inverse display"
```

ON BREAK and OFF BREAK Statements

Syntax

```
ON BREAK statementlist
OFF BREAK
```

Action

The ON BREAK statement is used in a program to control what action to take when a Control-C character (the break character) is typed. The ON BREAK statement is a special case of the ON ERR statement which is used to handle all other runtime errors. The ON BREAK statement must be executed to activate the break handling mechanism.

When a Control-C character is typed, and an ON BREAK statement has been executed, control will temporarily suspend and transfer to the sequence of statements following the ON BREAK. After the break has been handled, control may resume at the previous point by executing the RESUME statement.

The OFF BREAK statement cancels the ON BREAK statement. If no ON BREAK statement is active when a Control-C is typed, execution of the program terminates.

If the BREAK OFF statement has been executed, TML BASIC does not check for the Control-C character. Thus, it is impossible to invoke the user break mechanism to transfer control to the ON BREAK statement list or abort the program. BREAK ON turns the checking for Control-C back on.

See Also

BREAK ON and BREAK OFF
ON ERR
RESUME

Compiler/Interpreter Differences

The ON BREAK statement requires a significant amount of code to be generated by TML BASIC to implement this statement. Since most programs do not use the ON BREAK statement, TML BASIC allows you to turn off the code generation needed to support this statement. This is done by turning off the On Error option in the Preferences Dialog or by using the $OnError metastatement. If the On Error code generation is turned off and a program uses this statement, TML BASIC will give the error:

"On Error option must be ON for this Statement".

Further, you must instruct TML BASIC to generate code to check for the Control-C character. This is done by turning on the Keyboard Break option in the Preferences Dialog or with the $KeyboardBreak metastatement. If you use the ON BREAK statement (and $OnError is ON), but forget to turn on the Keyboard Break option, TML BASIC will give the error:

"Keyboard Break must be ON for this Statement".

Example

```
ON BREAK GOTO HandleBreak

Wait:
   PRINT "Wait for break"
   GOTO Wait

HandleBreak:
   PRINT "Break occurred"
   GET$ A$
   IF A$ = "." THEN END
   RESUME
```

ON EOF# and OFF EOF# Statements

Syntax

```
ON EOF# FileNumber StatementList
OFF EOF# FileNumber
```

Action

The ON EOF# statement allows a program to control what action to perform when an attempt is made to read a file past its end of file mark. If an ON EOF# statement has been previously executed when a read past the end of file occurs, program control unconditionally transfers to the sequence of statements after the ON EOF# statement. *FileNumber* must be the file reference number of an open file.

The OFF EOF# statement cancels the end of file trapping that was activated with the ON ERR# statement. If a program attempts to read past the end of file and no ON EOF# statement is active the standard TML BASIC error mechanism is used. That is, if an ON ERR statement is active, the sequence of statements associated with that statement is executed, otherwise, execution aborts.

Unlike the ON BREAK, ON KBD and ON ERR and ON EXCEPTION statements, program control unconditionally branches to the sequence of statements after the ON EOF#. You *cannot* use the RETURN or RESUME statements when handling the end of file error. The ON EOF# statement does not require the $OnError metastatement be ON.

See Also

EOF
EOFMARK
Chapter 9, Files

Example

```
OPEN SomeFile$, AS #1

ON ERR #1 GOTO EofEncountered

NextLine:
   INPUT #1; Line$
   PRINT Line$
   GOTO NextLine

EofEncountered:
   PRINT "EOF encountered for file "; EOF
   CLOSE #1
   END
```

ON ERR and OFF ERR Statements

Syntax

```
ON ERR Statementlist
OFF ERR
```

Action

The ON ERR statement is used in a program to control what action to perform when a runtime error occurs.

When a runtime error such as an "Overflow Error" or "Illegal Quantity Error" occurs, and an ON ERR statement has been executed, control will temporarily suspend and transfer to the sequence of statements following the ON ERR. After the error has been handled, control may resume at the previous point by executing the RESUME statement, or at the statement following the error by executing the RESUME NEXT statement.

The reserved variable ERR may be used in the sequence of statements handling the error in order to determine exactly what runtime error occurred and to respond accordingly.

If a program contains more than one ON ERR statement the most recently executed ON ERR statement is the one which receives control. A user break error (typing a Control-C) is handled separately by the ON BREAK statement.

See Also

ON BREAK
RESUME

Compiler/Interpreter Differences

The ON ERR statement requires a significant amount of code be generated by TML BASIC to implement this statement. Since many programs do not use the ON ERR statement, TML BASIC allows you to turn off the code generation needed to support this statement. This is done by turning off the On Error option in the Preferences Dialog or by using the $OnError metastatement. If the On Error code generation is turned off and a program uses this statement, TML BASIC will give the error:

"On Error option must be ON for this Statement".

Further, you should instruct TML BASIC to generate debugging code to check for runtime errors such as "Overflow Error", "Illegal Quantity Error", etc. This is done by turning on the Debug option in the Preferences Dialog or with the $Debug metastatement.

Example

```
ON ERR GOTO HandleError

x% = 20000
x% = x% + 15000
PRINT "x%=";x%
END

HandleError:
   IF ERR = 1 THEN          'An Overflow Error
      x% = 0
      RESUME
   ELSE
      PRINT "RUNTIME ERROR="; ERR
      END
   END IF
```

OUTPUT:

```
x%=15000
```

ON EXCEPTION and OFF EXCEPTION Statements

Syntax

```
ON EXCEPTION statementlist
OFF EXCEPTION
```

Action

The ON EXCEPTION statement is a separate version of the ON ERR statement for errors that occur in floating-point mathematical computations. TML BASIC implements floating-point operations using the built-in Standard Apple Numeric Environment (SANE) floating-point engine. SANE defines several error conditions which might occur while performing floating-point operations. They are

- Invalid operation (such as SQRT(-2))
- Overflow
- Underflow
- Divide by zero
- Unordered compare
- Inexact result

It is possible to define which of these errors are signaled to a TML BASIC program using the EXCEPTION ON statement.

The ON EXCEPTION statement is used in a program to control what action to take when the EXCEPTION ON statement has defined that certain floating-point errors should be signaled. The behavior of the ON EXCEPTION statement is exactly like the ON ERR statement. See the description of this statement for more information.

ON KBD and OFF KBD Statements

Syntax

```
ON KBD StatementList
OFF KBD
```

Action

The ON KBD statement is used to cause program control to automatically execute a sequence of statements whenever a keypress is detected at the keyboard.

After an ON KBD statement is executed, the program continues executing normally. But, as soon as a key is pressed, execution branches to the sequence of statements included in the ON KBD statement. Note that when the ON KBD statement is encountered during normal program execution, the statements following the reserved words ON KBD are not executed.

The branch to the ON KBD statement list is treated as a GOSUB to a subroutine. Therefore, the sequence of statements should end with a RETURN statement to continue normal program execution. The effect of executing the ON KBD statement is disabled after a keypress occurs. To re-enable it, the ON KBD statement must be executed again.

To disable the effect of ON KBD, execute the OFF KBD statement.

Note that when ON KBD is in effect, the program cannot be aborted using the Control-C character. This is because, the keypress is treated like any other keyboard character, and program control transfers to the sequence of statements after the ON KBD statement.

See Also

GOSUB
KBD

Example

```
ON KBD GOTO ShowKey                 'Activate keyboard event trapping

Wait: GOTO Wait                     'Infinite loop to wait for keypresses

ShowKey: PRINT "The key = ";KBD
         IF KBD = ASC(".") THEN END 'Quit when the period is pressed
         ON KBD GOTO ShowKey        'Reactivate keyboard event trapping
         RETURN
```

ON...GOSUB Statement

Syntax

```
    ON NumericExpression GOSUB Label {,Label}
```

Action

The ON...GOSUB statement is used to cause program control to branch to a subroutine based upon the value of a *NumericExpression*. After the reserved word GOSUB is a list of one or more labels separated by commas. The labels must designate subroutines which end with the RETURN statement.

The value of the *NumericExpression* determines which subroutine is executed. The value of *NumericExpression* must be an integer in the range 0 to 255. If the value is equal to one (1), control transfers to the subroutine designated by the first label, if the value is equal to two (2), control transfers to the subroutine designated by the second label, etc. If the value equals zero (0), or greater than the number of labels specified, the statement is ignored, and execution continues with the next statement.

See Also

GOSUB
ON...GOTO
Chapter 7, Labels
Chapter 8, Subroutines

Example

```
PRINT "Database options..."
PRINT "   1) Sort"
PRINT "   2) Print"
PRINT "   3) Enter record"
PRINT "   4) Delete record"
PRINT "   5) Quit"
PRINT "Enter selection: ";
GET$ Option$
Option% = ASC(Option$) - ASC("1") + 1
ON Option% GOSUB doSort,doPrint,doEnter,doDelete,doQuit
```

ON...GOTO Statement

Syntax

```
    ON NumericExpression GOTO Label {,Label}
```

Action

The ON...GOTO statement is used to cause program control to branch to a label based upon the value of a *NumericExpression*. After the reserved word GOTO is a list of one or more labels separated by commas.

The value of the *NumericExpression* determines to which label execution transfers. The value of *NumericExpression* must be an integer in the range 0 to 255. If the value is equal to one (1), control transfers to the statement designated by the first label, if the value is equal to two (2), control transfers to the statement designated by the second label, etc. If the value equals zero (0), or greater than the number of labels specified, the statement is ignored, and execution continues with the next statement.

See Also

GOTO
ON...GOSUB
Chapter 7, Labels

Example

```
PRINT "Database options..."
PRINT "   1) Sort"
PRINT "   2) Print"
PRINT "   3) Enter record"
PRINT "   4) Delete record"
PRINT "   5) Quit"
PRINT "Enter selection: ";
GET$ Option$
Option% = ASC(Option$) - ASC("1") + 1
ON Option% GOTO doSort,doPrint,doEnter,doDelete,doQuit
```

ON TIMER and OFF TIMER Statements

Syntax

```
ON TIMER(Seconds) StatementList
OFF TIMER
```

Action

The ON TIMER enables event trapping using the 1-second interrupt capability of the Apple IIGS clock. *Seconds* is an integer expression that sets a countdown interval of the value of *Seconds*. *Seconds* must be in the range 2 to 86400.

When the interval counter reaches zero, the countdown is complete, and execution branches (like a GOSUB) from the currently completed program statement, to the ON TIMER *StatementList*. *StatementList* must end with a RETURN statement to return control to the next sequential statement in the program.

The TIMER countdown is approximate only and does not guarantee a precise amount of time. Some higher priority operations, such as disk I/O or AppleTalk communications might even lock out the timer interrupt for more than a second. The ON TIMER statement will have no effect unless the 1-second interrupt is enabled by the TIMER ON statement.

OFF TIMER disables the most recently executed ON TIMER statement.

See Also

GOSUB
RETURN
TIMER ON

OPEN Statement

Syntax

```
    OPEN Pathname, [ FILTYP= DIR|TXT|SRC|BDF|Filetype ]
        [ FOR INPUT|OUTPUT|APPEND|UPDATE ] AS # Filenumber [, Recordsize ]
```

Action

The OPEN statement is used to open files for access, and must precede any file I/O routines accessing a given file. The minimum required arguments following the reserved word OPEN are the file's pathname followed by a comma, the reserved word AS and a file reference number. The file must have been previously created and must exist on a disk currently mounted in a disk drive. If a partial pathname is used, it is joined with prefix 0 to create the full pathname. The file reference number is used in all subsequent TML BASIC I/O statements for accessing the file.

The optional FOR clause in the OPEN statement is used to qualify the *access mode* for the file. The supported access modes are INPUT, OUTPUT, APPEND and UPDATE. If the FOR clause is not used, the file is opened for UPDATE. The FOR INPUT clause specifies that the file is opened for read-only access, and cannot be written to. For example:

```
    OPEN myFile$, FOR INPUT AS #10
```

The FOR OUTPUT clause specifies that the file is opened for write-only access, and cannot be read from. For example:

```
    OPEN myFile$, FOR OUTPUT AS #10
```

The FOR APPEND option is a variant of the FOR OUTPUT clause. It is used for sequential access (discussed later) to allow the PRINT# and WRITE# statements to append new information to the end of a file without disturbing any existing data in the file. For example:

```
    OPEN myFile$, FOR APPEND AS #10
```

Finally, the FOR UPDATE clause is used to open a file for read-write access so long as the filetype supports such access. For example, you can't read from a printer.

The optional FILTYP= clause of an OPEN statement is used to specify the type of file. The FILTYP= clause is primarily used to ensure that a file being opened is of the expected filetype. If a program attempts to open a file using the FILTYP= clause and the file's type does not match the specified filetype, the file will not be opened and an error message wil be reported. Any of the predefined filetype names (see CREATE) can be used with the FILTYP= clause or an unsigned integer value.

The FILTYP= clause is also used with the OPEN statement to open files which have not been created. If the OPEN statement finds that the specified file does not exist, and the FILTYP= clause is given, it will implicitly call the CREATE statement first and then open the file.

Finally, the optional *Recordsize* argument is used to specify the record size for a random access to the file using the INPUT# and GET# statements for non-Basic Data Files. If the file being opened is an existing BASIC Data File, the record size argument is ignored and the record size used is the size specified when the file was created.

See Also

CLOSE
Chapter 9

Example

```
OPEN "HELLOWORLD.BAS", AS #10
OPEN "/TML/MYSTUFF/INVOICES", FOR INPUT AS #20
OPEN aFile$, FOR UPDATE AS #20, 100
OPEN ".PRINTER", AS #1
OPEN ".MODEM", AS #2
```

OUTPUT# Statement

Syntax

```
    OUTPUT  #FileNumber
```

Action

The OUTPUT# statement is used to redirect output which is normally directed to the Apple IIGS text screen to a file previously opened with *FileNumber* as its file reference number. Recall that devices can be opened with a file reference number, thus allowing output to be redirected to devices as well. The printer is an example of a device.

The PRINT and CATALOG statements are the only statements affected by the OUTPUT# statement. To restore output back to the text screen use the statement OUTPUT #0.

See Also

CATALOG
OPEN
PRINT

Example

```
OPEN SomeFile$, AS #1
OUTPUT #1

PRINT "The following are the files on my disk"
PRINT
CATALOG

OUTPUT #0
CLOSE #1
END
```

PDL Function
PDL9 Reserved Variable

Syntax

```
    PDL(NumericExpression)
    PDL9
```

Action

The PDL function reads the position of the game control paddle and returns its position as an integer value in the range 0 to 255.

The PDL function actually reads the position of the paddle twice as fast as the original Apple II routines and discards the least significant bit, thus eliminating the uncertainty caused by the variable processor speed of the Apple IIGS. The reserved variable PDL9 returns the 9-bit result calculated by the prior execution of the PDL function.

NOTE: Reading any paddles in quick succession will tend to produce unstable results because of the hardware coupling among all four paddles. Using the JOYX function will eliminate this interaction when reading two paddles of both axes of a joystick.

Example

```
ReadPaddles: PRINT PDL(0), PDL9
             GOTO ReadPaddles
```

PEEK Function

Syntax

```
    PEEK(NumericExpression)
```

Action

The PEEK function reads a byte from the memory address specified by *NumericExpression* and returns an integer value in the range 0 to 255 inclusive. The *NumericExpression* must be a positive integer less than 2^24 and must represent a legal Apple IIGS memory address.

The PEEK function should only be used in special circumstances in a program. You should exercise great care when using PEEK not to read memory mapped I/O devices and control registers, since merely reading those addresses can cause unpredictable side effects, including system crash.

Programmers concerned about writing programs that will run on new versions of the Apple IIGS product family should avoid the use of the PEEK function with addresses that might not be compatible with future machines or system software.

See Also

POKE

Example

```
'Use hard coded address to read the Option key sense input
OptionKey% = PEEK(14729314)    'Hex address E0C062
PRINT OptionKey%

'Use variable address to read a byte of memory in a variable
myStr$ = "TML BASIC"
Address@ = VARPTR$(myStr$)
PRINT CHR$(PEEK(Address@+1))  'Display first letter of string in memory
```

OUTPUT:

```
127
T
```

PFX$ Function

Syntax

```
    PFX$(Prefix)
```

Action

The PFX$ function returns a string which is the current value of the indicated ProDOS 16 prefix. *Prefix* must be a numeric expression in the range 0 through 8 inclusive, otherwise an "Illegal Quantity Error" results. The prefix values 0 through 7 return the ProDOS 16 prefix by that number, the prefix value 8 returns the pseudo-prefix equal to the boot volume name.

See Also

PREFIX
PREFIX$
Chapter 9 - Files

Example

```
'Print each of the ProDOS 16 prefixes
FOR i% = 0 to 8
   PRINT PFX$(i%)
NEXT i%
```

PI Reserved Variable

Syntax

```
    PI
```

Action

PI is a reserved variable whose value is π, accurate to 20 decimal digits. The value of PI is stored as a SANE extended precision real value in order to provide the greatest amount of accuracy possible in expressions. TML BASIC automatically converts PI to any numeric type on assignment, with of course a loss of accuracy.

Example

```
Radians = Degrees * PI / 180   'Convert degrees to radians
```

POKE Statement

Syntax

```
    POKE NumericExpression, Value
```

Action

The POKE statement writes a byte *Value* to the memory address specified by *NumericExpression*. *Value* must be in the range 0 to 255 or an "Illegal Quantity Error" occurs. The *NumericExpression* must be a positive integer less than 2^24 and must represent a legal Apple IIGS address.

The POKE statement should only be used in special circumstances in a program. You should exercise great care when using the POKE statement not to accidentally write to memory mapped I/O devices, control registers or other addresses not allocated to your program.

Programmers concerned about writing programs that will run on new versions of the Apple IIGS product family should avoid the use of the POKE statement with addresses that might not be compatible with future machines or system software.

See Also

PEEK

Example

```
anInt% = 0
PRINT anInt%
POKE VARPTR(anInt%),2        'Assignment the HARD way!
POKE VARPTR(anInt%)+1,1
PRINT anInt%
```

OUTPUT:

```
0
258
```

POP Statement

Syntax

```
    POP
```

Action

The POP statement is used to jump out one nested subroutine level by removing the subroutine "return address" from the TML BASIC *Runtime Stack*. Thus, when the next RETURN statement is executed, instead of branching back to the statement after the most recently executed GOSUB statement, control branches to the statement after the second most recently executed GOSUB statement.

If a POP is executed in a program without having executed a GOSUB statement the "RETURN/POP without matching GOSUB" error occurs.

See Also

GOSUB
RETURN

Example

```
Print "Start Program"
GOSUB First
PRINT "End Program"
END

First:
   PRINT "Enter subroutine First"
   GOSUB Second
   PRINT "Leave subroutine First"
   RETURN

Second:
   PRINT "Enter subroutine Second"
   POP
   PRINT "Exit subroutine Second"
   RETURN
```

OUTPUT:

```
Start Program
Enter subroutine First
Enter subroutine Second
Exit subroutine Second
End Program
```

PREFIX Statement

Syntax

```
    PREFIX DirectoryPath

    PREFIX PrefixNum, DirectoryPath
```

Action

The PREFIX statement is used to set a ProDOS prefix. The first form of the statement sets the ProDOS prefix 0 to the pathname specified by *DirectoryPath*. The second form of the statement sets any prefix numbered 0 through 7 as specified by the *PrefixNum* argument. If the pathname used in the PREFIX statement is illegal, the "Bad Path Error" occurs.

Compiler/Interpreter Differences

GS BASIC provides variations of the PREFIX statement that display the current values of prefixes. This is not supported in TML BASIC.

See Also

Chapter 9

Example

```
PREFIX "/TML/PART1.EXAMPLES"
PREFIX 4,"/TML/MYWORK/NDA"
```

PREFIX$ Modifiable Reserved Variable

Syntax

```
    PREFIX$
```

Action

PREFIX$ is a modifiable reserved variable whose value is the ProDOS 16 default prefix. That is, prefix zero (0). If a new value is assigned to the reserved variable, the ProDOS 16 prefix zero is changed to reflect the new pathname. If an illegal ProDOS 16 pathname is assigned to PREFIX$, the "Bad Path Error" occurs.

See Also

PFX$
PREFIX
Chapter 9 - Files

Example

```
PRINT "The current default prefix = "; PREFIX$

INPUT "Enter a new default prefix: "; NewPrefix$
PREFIX$ = NewPrefix$
```

PRINT Statement

Syntax

```
    PRINT { [,|;] [AnyExpression ] } [,|;]
```

Action

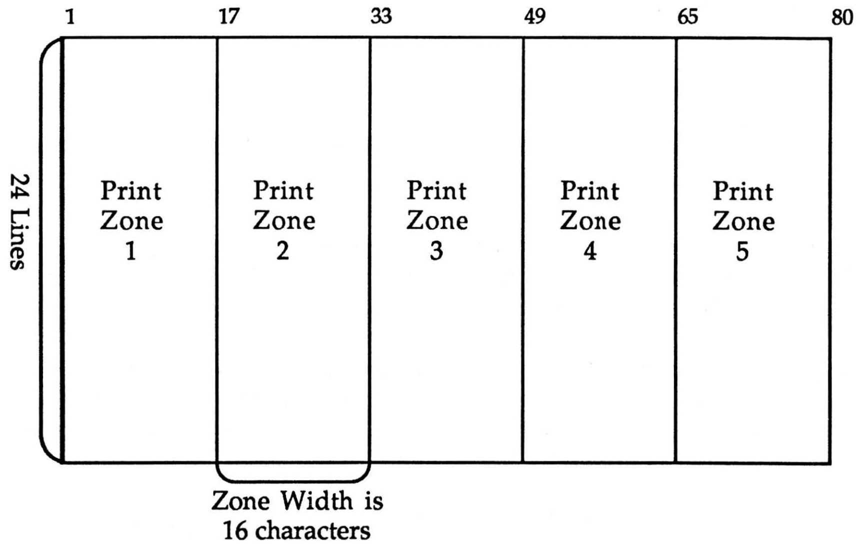
The PRINT statement displays text on the Apple IIGS text screen. The PRINT statement is used to print the values of numeric and string expressions. The PRINT statement may contain any number of expressions separated by either a comma or semicolon. Each expression is called a *print item*. Actually, multiple expressions can be separated by spaces, but it is good programming practice to use either a comma or a semicolon so it is clearly understood that more than one expression is included within the PRINT statement.

When a string expression appears in a PRINT statement, the exact characters in the string are displayed to the text screen at the current text location (the location of the cursor). When a numeric expression is printed, the binary representation of the numeric value is first converted to a string and then displayed at the current text location. The conversion is controlled by the SHOWDIGITS reserved variable. If the numeric expression contains an integer value, it is displayed as an integer unless SHOWDIGITS is too small, in which case the number is displayed in scientific notation.

When using the semicolon as a separator between multiple expressions in a PRINT statement, TML BASIC positions the cursor immediately following the last character displayed. Thus, the next expression is displayed adjacent to the previous print item. Using a comma as a separator causes TML BASIC to perform a tab operation before the next print item is displayed. The tab width of the PRINT statement is 16 characters. The spaces between each tab is called a *print zone*. The diagram on the next page illustrates how the 80 columns of a text screen are divided into five print zones.

After all print items in a PRINT statement have been displayed, the text cursor is moved to the first column of the next line. If the cursor is on the last line of the screen, the entire contents of the screen is scrolled up one line. Thus, a PRINT statement containing zero will display a blank line.

In some cases, a program may not want the PRINT statement to advance the text location to the next line after it has displayed all of its print items. Whenever a PRINT statement ends with a comma or a semicolon, the PRINT statement will not advance to the next line.



The PRINT statement also works in the Super Hi-Res graphics screen. When the PRINT statement is executed, it examines the current screen mode. If the screen is in text mode (the default), text is displayed in the normal fashion. However, if the screen is in graphics mode, text is displayed to the current *GrafPort* (window) using QuickDraw graphics routines. The text is drawn beginning at the current location of the QuickDraw pen. None of the TML BASIC screen position commands work in the graphics screen. To move the pen, QuickDraw commands such as Move and MoveTo must be used. For more information about QuickDraw see Chapter 12.

See Also

PRINT USING
PRINT#
SHOWDIGITS
SPC
TAB

Example

```
PRINT "The average of three numbers is "; (43 + 27 + 23) / 3

FOR i% = 1 TO 5
   PRINT SPACE$(i%); i%
NEXT i%

PRINT tbl1,tbl2,tbl3
```

PRINT USING Statement

Syntax

```
    PRINT USING UsingSpecification [; Expression {, Expression } ] [;]
```

Action

The PRINT USING statement is an advanced form of the PRINT statement. The PRINT USING statement contains the special *UsingSpecification* which controls the format of the individual print items displayed to the text screen.

The *UsingSpecification* may be a string variable, string constant or a label which contains an IMAGE statement. In each case, the the information is expressed in the same way. See the descrition of the IMAGE statement for a complete list of the formatting specifications available.

In the PRINT USING statement, the print items (*Expressions*) are separated by commas. The commas do not cause a tab action to the next print zone as they do in the PRINT statement since formatting is controlled by the *UsingSpecification*. The trailing semicolon can still be used, however, preventing the PRINT statement from advancing to the next line.

See Also

PRINT
IMAGE

PRINT# Statement

Syntax

```
PRINT# FileNumber [, RecordNumber ] [; Expression {,|; Expression }] [;]
```

Action

The PRINT# statement writes a line of text to a file in the same way that the PRINT statement does to the screen. The reserved word PRINT# is followed by the file reference number of an open file to write to, a semicolon, and a list of expressions separated by commas or semicolons.

PRINT# automatically performs any necessary numeric to string type conversions before writing to the file. Numeric values are formatted using the same rules as the PRINT statement. That is, SHOWDIGITS controls the format of numbers generated by PRINT#. Using the comma as the separator between expressions causes the tab action to the next print zone, while the semicolon does not. The SPC and TAB functions can be used as well.

An optional form of the PRINT# statement permits random access to a text file. To perform random access using the PRINT# statement, include a record number after the file reference number. Recall that the file must be opened using the OPEN statement with the optional record size argument specified to define the size of a record in the text file. Consider the following statements:

See Also

PRINT USING
PRINT
SHOWDIGITS
SPC
TAB
Chapter 9

Example

```
PRINT #10; anInt%, aReal, aStr$  'Sequentially write several values to a file

OPEN "AFILE", AS #10, 20         'Open a file for random access
PRINT #10,6; aLine$              'Write a line of text at random record 6
```

PRINT# USING Statement

Syntax

```
PRINT# FileNumber [, RecordNumber ] USING UsingSpecification
    [; Expression {,|; Expression }] [;]
```

Action

The PRINT# USING statement is an advanced form of the PRINT# statement. The PRINT# USING statement contains the special *UsingSpecification* which controls the format of the individual print items written to a text file.

The *UsingSpecification* may be a string variable, string constant or a label which contains an IMAGE statement. In each case, the the information is expressed in the same way. See the descrition of the IMAGE statement for a complete list of the formatting specifications available.

In the PRINT# USING statement, the print items (*Expressions*) are separated by commas. The commas do not cause a tab action to the next print zone as they do in the PRINT# statement since formatting is controlled by the *UsingSpecification*. The trailing semicolon can still be used, however, preventing the PRINT# statement from advancing to the next line.

PUT# Statement

Syntax

```
    PUT# FileNumber [,[Length] [,RecordNumber]];StructureVariable
```

Action

The PUT# statement writes a number of bytes from a structure array to a binary file. The reserved word PUT# is followed by the file reference number of an open binary file to write to, a semicolon, and a structure array variable reference (includes a subscript). The number of bytes transferred is equal to the record size of the file.

Using the optional *Length* argument in the PUT# statement, it is possible to override the number of bytes transferred to some value other than the record size. The PUT# statement can also be used for random assess using the optional *RecordNumber* argument.

See Also

OPEN
GET#
Chapter 9, Files

Example

```
DIM  myData!(11)

'Open a binary file whose record size is 4
OPEN "SOMEFILE", FILTYP=0 FOR OUTPUT AS #1, 4

PUT #1; myData!(0)                  'Write 4 bytes to first record in file
PUT #1,,3; myData!(4)               'Write 4 bytes starting at record 3
PUT #1,2,5; myData!(0)              'Write 2 bytes starting at record 5

CLOSE #1
```

R.STACK Functions

Syntax

```
R.STACK%(NumericExpression)
R.STACK@(NumericExpression)
R.STACK&(NumericExpression)
```

Action

The R.STACK functions return data from the CALL return stack. The CALL return stack is a 32 byte (16 words) buffer used by the CALL, CALL% and EXFN_ statements for storing the values returned by an Apple IIGS Toolbox routine.

Because each toolbox routine returns a variable amount of information in different data types, the CALL return stack can be accessed for integer, double integer and long integer values. The *NumericExpression* parameter is a *word* offset into the stack. The number of bytes read from the stack beginning at that point depends upon which R.STACK function is called. R.STACK% returns an integer value reading 2 bytes of data from the stack; the R.STACK@ function returns a double integer reading 4 bytes of data from the stack; and finally, R.STACK& returns a long integer reading 8 bytes of data from the stack. Thus, R.STACK% may be indexed in the range 0 to 16, R.STACK@ in the range 0 to 15, and R.STACK& in the range 0 to 13.

R.STACK%(0) returns the *error code* returned by the toolbox routine. If the value is zero, then no error occurred. If the value is non-zero, an error occurred during the execution of the toolbox routine, and your program should take appropriate action.

R.STACK%(1) is the first word of data returned on the CALL stack.

See Also

CALL
CALL%
EXFN_
Chapter 11

Example

```
CALL NewHandle(1024,myMemoryID%,0,0)

IF R.STACK%(0) = 0 THEN
   myHandle@ = R.STACK@(1)
ELSE
   PRINT "Unable to allocate memory handle, error: ";R.STACK%(0)
END IF
```

RANDOMIZE Statement

Syntax

```
    RANDOMIZE NumericExpression
```

Action

Reseeds the random number generator with the value of *NumericExpression* as the new seed. *NumericExpression* must be in the range 1 to 2^31-2. Good values to use as a seed are values from the TIME function or the SECONDS@ reserved variable after the TIMER ON statement has been executed.

See Also

SECONDS@
TIME
TIMER ON

Example

```
RANDOMIZE 8849391
RANDOMIZE SECONDS@
RANDOMIZE TIME(2)*60+TIME(3)
```

READ Statement

Syntax

```
    READ VariableName {,VariableName}
```

Action

The READ statement assigns into one or more variables, values obtained from a program's DATA statements. The values are read beginning at the current DATA *list pointer*. The DATA list pointer initially points to the first constant in the first DATA statement of the program. The list pointer advances as values are read. It can also be changed to point to any DATA statement using the RESTORE statement.

If a READ statement attempts to assign a string data element to a numeric variable, a "Type Mismatch Error" occurs.

See Also

DATA
RESTORE

Example

```
READ A$,B$

RESTORE Names
READ C$,D$

PRINT A$,B$,C$,D$
END

Names: DATA Apple, Orange
       DATA Pear, Grape
```

OUTPUT:

```
Apple     Orange    Apple     Orange
```

READ# Statement

Syntax

```
    READ# FileNumber [, RecordNumber ] [; VariableName {, VariableName }]
```

Action

The READ# statement reads information from a BASIC Data File (BDF) into one or more variables. The reserved word READ# is followed by the file reference number of an open BDF file to read from, a semicolon, and a list of variables separated by commas.

If a READ# statement contains a numeric variable, the value at the current file position in the BDF file must also be a numeric value. If the file contains a string value, the "Type Mismatch Error" occurs. If the file does contain a numeric value, but its type does not match the variable in the READ# statement, the value is converted using the same rules as the CONV functions. Thus, it is possible the conversion will lose precision or even cause an "Overflow Error". If the READ# statement contains a string variable, the value at the current file position must be a string value, otherwise a "Type Mismatch Error" occurs.

An optional form of the READ# statement permits random access to a BDF file. To perform random access using the READ# statement include a record number after the file reference number.

See Also

WRITE#
Chapter 9

Example

```
READ #10; anInt1%, anInt2%, anInt3%    'Sequential access of a BDF file
READ #10,3; aStr$, aDblInt@            'Random access of a BDF file
```

REC Function

Syntax

```
    REC(FileNumber)
```

Action

The REC function returns the current record number of the file previously opened with its file reference number equal to *FileNumber*.

When using INPUT# or READ# statements to access the catalog of a directory, REC will return the number of the line currently being accessed.

See Also

OPEN
INPUT#
READ#

Example

```
OPEN "SOMEFILE", AS #1

FOR i% = 1 TO 5
   READ# 1,i%; myInt%
   PRINT "Record "; REC(1); " has integer value "; myInt%
NEXT i%

CLOSE #1
```

REM Statement

Syntax

```
    REM AnyText
```

Action

The REM statement, also called the *remark* statement, is used to place descriptive information about your code in a program. The REM statement continues to the end of the current line. It is not possible to follow the REM statement with another statement on the same line separated by a colon. When compiling a program, TML BASIC ignores the REM statement so that it has no effect on the program.

TML BASIC offers an alternative to the REM statement called the *Comment*. A comment behaves just like a REM statement, but consists only of the single quote (') character.

See Also

Chapter 7, Comments

Example

```
REM The following lines show how REM and ' can be used

Interest = Principle * Rate    : REM Calculate the interest due
Interest = Principle * Rate    'Calculate the interest due
```

RENAME Statement

Syntax

```
    RENAME OldPathname, NewPathname [,FILTYP= TXT|SRC|BDF|FileType]
```

Action

The RENAME statement is used to change the name of a volume, subdirectory or any other file. The arguments *OldPathname* and *NewPathname* must be string expressions which represent legal ProDOS 16 pathnames. The *OldPathname* must be the pathname of an existing file which is given the new pathname specified by *NewPathname*. Using RENAME, it is possible to change the local name of a file or to move the file to another subdirectory, but it is not possible to move the file to another disk by merely changing its name.

When the optional FILTYP= argument is used, the file type of *NewPathname* will be changed after the file is successfully renamed. It is possible only to change the file type of a file by using the FILTYP= where the value of *OldPathname* is the same as the *NewPathname*.

See Also

Chapter 9, Files

Example

```
'Make current directory be the PART1.EXAMPLES folder on the TML BASIC disk
PREFIX "/TML/PART1.EXAMPLES"

'Rename the HELLOWORLD.BAS file to HELLO.BAS
RENAME "HELLOWORLD.BAS", "HELLO.BAS"

'Rename HELLO.BAS so that it is now in the PART2.EXAMPLES folder
RENAME "HELLO.BAS", "/TML/PART2.EXAMPLES/HELLO.BAS"
```

REP$ Function

Syntax

```
    REP$(StringExpression,NumericExpression)
```

Action

The REP$ function returns a string containing a number of characters equal to *NumericExpression* whose characters are all equal to the first character of *StringExpression*.

NumericExpression must be an integer in the range 1 to 255 inclusive or an "Illegal Quantity Error" will occur. If the value of *StringExpression* is a null string then REP$ returns a string of question mark characters (?).

See Also

SPACE$

Example

```
Msg$ = "TML BASIC"
PRINT Msg$
PRINT REP$("-",LEN(Msg$))
PRINT

PRINT REP$("",5)
```

OUTPUT:

```
TML BASIC
---------
?????
```

RESTORE Statement

Syntax

```
    RESTORE [Label ]
```

Action

The RESTORE statement is used to move TML BASIC's *DATA list pointer* to the first data item in the DATA statement indicated by *Label*. After the RESTORE statement is executed, the next READ statement will begin reading values starting with the indicated DATA statement. If a *Label* is not given in the RESTORE statement, the next READ will begin reading from the first DATA statement in the program. Using this statement, a DATA statement can be read and re-read as many times as a program needs.

If the line indicated by *Label* does not contain a DATA statement then the result of the next READ statement is unpredictable.

See Also

DATA
READ

Example

```
READ A$,B$

RESTORE Names
READ C$,D$

PRINT A$,B$,C$,D$
END

Names: DATA Apple, Orange
       DATA Pear, Grape
```

OUTPUT:

```
Apple     Orange    Apple     Orange
```

RESUME Statement

Syntax

```
RESUME
RESUME NEXT
```

Action

The RESUME statements restart execution of a program after error handling was trapped by the ON BREAK, ON ERR or ON EXCEPTION statements.

The RESUME statement causes execution to restart with the statement that caused the error. The RESUME NEXT statement causes execution to restart with the statement immediately following the statement which caused the error.

If the RESUME statement is executed when the program has not encountered an error, it has no effect, and execution continues with the next statement.

See Also

ON BREAK
ON ERR
ON EXCEPTION

Compiler/Interpreter Differences

The RESUME statement requires a significant amount of code be generated by TML BASIC to implement this statement. Since most programs do not use the ON ERR...RESUME statements, TML BASIC allows you to turn off the code generation needed to support this statement. This is done by turning off the On Error option in the Preferences Dialog or by using the $OnError metastatement. If the On Error code generation is turned off and a program uses this statement, TML BASIC will report the error: "On Error option must be ON for this Statement".

Example

```
ON BREAK GOTO HandleBreak

Wait:
   PRINT "Wait for break"
   GOTO Wait
HandleBreak:
   PRINT "Break occurred"
   GET$ A$
   IF A$ = "." THEN END
   RESUME
```

RETURN Statements

Syntax

```
RETURN
RETURN 0
```

Action

RETURN causes program execution to branch to the statement after the most recently executed GOSUB instruction.

When a GOSUB statement is executed, TML BASIC stores the address of the statement following the GOSUB statement on the *Runtime Stack*. When the RETURN statement is executed, the address on the Runtime Stack is removed, and control is transferred to that address. If the RETURN statement is executed without having first executed a GOSUB statement, the runtime error "RETURN/POP without matching GOSUB"is reported.

RETURN 0 is a special case of RETURN statement used for event-handling subroutines defined by EVENTDEF and MENUDEF. These subroutines are implicitly called by the TASKPOLL statement. This special form of the RETURN statement is required because of the different calling mechanism used by the TASKPOLL statement. As such, the RETURN 0 statement should never be used by a subroutine which is called by a normal GOSUB statement.

See Also

GOSUB
EVENTDEF
MENUDEF
POP
TASKPOLL

Example

```
MainProgramStart:  PRINT "Main program "
                   GOSUB MySubroutine
                   PRINT "Main program again"
                   END
MySubroutine:      PRINT "Hi from MySubroutine"
                   RETURN
```

OUTPUT:

```
Main program
Hi from MySubroutine
Main program again
```

RIGHT$ Function

Syntax

```
    RIGHT$(StringExpression, NumericExpression)
```

Action

The RIGHT$ function returns the *NumericExpression* string of characters occurring rightmost in the string *StringExpression*.

StringExpression may be any string variable, string constant or string expression. If *NumericExpression* is a real value, it is rounded to the nearest whole number. The value of *NumericExpression* must be between 1 through 255 inclusive or an "Illegal Quantity Error" occurs. To find the number of characters in the string, use the LEN function.

See Also

LEFT$
LEN
MID$

Example

```
PRINT RIGHT$("TML BASIC is great",5)
```

OUTPUT:

```
great
```

RND Function

Syntax

```
    RND(NumericExpression)
```

Action

The RND function returns a random, real value, between 0 and 1.

Numbers generated by RND are not actually random, but are the result of a pseudo-random algorithm to a starting seed value. Given the same seed value, RND will produce the same sequence of "random" numbers. To set the seed value use the RANDOMIZE statement.

Calling RND with *NumericExpression* equal to zero (0) returns the previous random number, any other value returns the next "random" number in the sequence.

See Also

RANDOMIZE

Example

```
dummy% = TIME(0)         'Read the Apple IIGS clock
RANDOMIZE TIME(3)        'RANDOMIZE given the current seconds

FOR i = 1 to 5           'Compute 5 random numbers
   PRINT RND(i)
NEXT i
```

OUTPUT:

```
0.6561249
0.4910289
0.7219557
0.9089912
0.415245
```

ROUND Function

Syntax

`ROUND(`*`NumericExpression`*`)`

Action

The ROUND function returns the integer value nearest the value of *NumericExpression*. ROUND should be used in place of the commonly used INT(*NumericExpression* + 0.5), since it returns a result consistent with other SANE capabilities.

See Also

INT
Chapter 7

Example

```
FOR i = 1 TO 2 STEP 0.1
   PRINT i,ROUND(i)
NEXT i
```

OUTPUT:

```
1          1
1.1        1
1.2        1
1.3        1
1.4        1
1.5        2
1.6        2
1.7        2
1.8        2
1.9        2
```

RUN Statement

Syntax

```
RUN PathName
```

Action

The RUN statement is used to directly execute another program from the current program without having to return to the Apple IIGS Finder. When the RUN program terminates, control returns to the Finder. To return control back to the calling program use the CHAIN statement.

PathName must be a string expression which is a legal ProDOS 16 pathname for an executable program. The pathname can be the name of any compiled TML BASIC program or any other application you might own.

Compiler/Interpreter Differences

TML BASIC does not allow the optional *Label* argument that GS BASIC supports. When the RUN statement is executed, TML BASIC begins execution of the next program at its beginning.

See Also

CHAIN
Chapter 9, Files

Example

```
'Ask user for the next program to run
INPUT "Enter the name of the program you wish to run: "; ProgName$

'Now run the requested program
RUN ProgName$
END
```

SCALB Function

Syntax

```
    SCALB(Scale, NumericExpression)
```

Action

The SCALB function scales the *NumericExpression* by 2^*Scale*. The function effectively shifts the value of *NumericExpression* right or left *Scale* binary digits.

LOGB is related to SCALB, returning the *Scale* for a given *NumericExpression*.

See Also

LOGB

Example

```
PRINT SCALB(4,12)     'Equivalent to 2^4 * 12
```

OUTPUT:

```
192
```

SCALE Function

Syntax

```
    SCALE(Scale, NumericExpression)
```

Action

The SCALE function is used in conjunction with the PRINT USING statement. SCALE converts the *NumericExpression* argument to its string representation and then shifts the decimal point to the right *Scale* number of digits. If the value of *Scale* is positive then the decimal point is moved to the right, otherwise the decimal point is moved to the left.

See Also

PRINT USING

Example

```
A&  = 12345678901234567
PRINT USING "$$20&#.##";SCALE(-2,A&)

OUTPUT:

$123,456,789,012,345.67
```

SECONDS@ Reserved Variable

Syntax

```
    SECONDS@
```

Action

The SECONDS@ reserved variable contains the value of a counter maintained by the TIMER ON statement. SECONDS@ returns a double integer value in the range -1 through 86400. The value zero is returned until a TIMER ON statement has been executed. If TIMER OFF mode is currently in effect, the value of SECONDS@ does not change.

Due to the presence of numerous interrupt sources in the Apple IIGS, many of which have higher priority than the 1-second clock interrupt, the SECONDS@ value is not always exact. However, SECONDS@ will always be exact immediately after execution of the TIMER ON statement. TIMER ON may be used as often as needed during a program.

See Also

TIMER ON and TIMER OFF
ON TIMER

Example

```
TIMER ON
RANDOMIZE SECONDS@
```

SET Statement

Syntax

```
SET (StructureArrayReference [, Size] ) = NumericExpression
SET (StructureArrayReference [, Size] ) = ^StringVariable
SET (StructureArrayReference [, Size] ) = *divar[,Length]
```

Action

The SET statement is used to store a variable or an expression into a structure array. There are three different forms of the SET statement, each discussed separately below.

The `StructureArrayReference` must be a structure array variable which has been previously declared. The optional *Size* argument determines the number of bytes transferred into the structure array variable. The value of *Size* must be a positive integer greater than or equal to one (1), but not larger than the size of the structure array. If the *Size* argument does not appear, the number of bytes transferred is the size of the expression after the equal sign.

The first form of the SET statement assigns the value of *NumericExpression* into the structure array variable. To ensure the type and size of the *NumericExpression* value, the CONV functions can be used. See Chapter 7 for a complete description of the TML BASIC numeric types and their respective sizes.

The second form of the SET statement is used to store a TML BASIC string into a structure array as a counted string (Pascal string). *StringExpression* can be a string expression or a string variable. A length byte is stored in the first specified element of the structure array followed by the elements of the string expression.

The third form of the SET statement allows two types of direct memory assignment to a structure array. When the *Length* expression is omitted, the value of the double integer variable is used as the memory address of a 1 byte count, followed by 0 to 255 characters of string data that are assigned to the field in the structure array. If the *Length* expression is present, the address used is the address of length bytes of data. The count byte or the length parameter may be zero, and the length may be up to 32767.

See Also

DIM
VAR
Chapter 7

SGN Function

Syntax

```
    SGN(NumericExpression)
```

Action

The SGN function is used to determine the sign of a *NumericExpression*. SGN returns the integer value -1 if *NumericExpression* is negative, zero if *NumericExpression* is equal to 0, and 1 if *NumericExpression* is positive.

Example

```
PRINT SGN(-1234)
PRINT SGN(0)
PRINT SGN(5342)
```

OUTPUT:

```
-1
0
1
```

SHOWDIGITS Modifiable Reserved Variable

Syntax

```
    SHOWDIGITS
    SHOWDIGITS = NumericExpression
```

Action

The SHOWDIGITS modifiable reserved variable controls how many significant digits are displayed for numeric values by the PRINT statement.

The default value of SHOWDIGITS is 7, the number of significant digits in a single-precision real number. SHOWDIGITS can be set to integer values in the range 2 through 28.

SHOWDIGITS only effects the behavior of the PRINT statement and not the PRINT USING statement.

See Also

PRINT
PRINT USING

Example

```
FOR i% = 2 to 7
   SHOWDIGITS = i%
   PRINT PI
NEXT i%
```

OUTPUT:

```
3.1
3.14
3.141
3.1416
3.14159
3.141593
```

SIN Function

Syntax

```
    SIN(NumericExpression)
```

Action

Returns the trigonometric sine of *NumericExpression*. *NumericExpression* is an angle expressed in radians. To convert radians to degrees, multiply by 180/π. To convert degrees to radians, multiply by π/180.

See Also

ATN
COS
PI
TAN

Example

```
PRINT "Sine of 45 degrees = '; SIN(45 * PI/180)
```

OUTPUT:

```
0.7071068
```

SPACE$ Function

Syntax

```
    SPACE$(NumericExpression)
```

Action

The SPACE$ function returns a string containing a number of spaces equal to *NumericExpression*.

NumericExpression must be an integer in the range 0 to 255 inclusive or an "Illegal Quantity Error" will occur.

See Also

REP$

Example

```
FOR i% = 0 to 5
   PRINT SPACE$(i%),"X"
NEXT i%
```

OUTPUT:

```
X
 X
  X
   X
    X
     X
```

SPC Function

Syntax

```
    SPC(NumericExpression)
```

Action

The SPC function is used to skip *NumericExpression* spaces after the last printed character in a PRINT or PRINT# statement.

NumericExpression must be an integer value in the range 0 through 255, otherwise an "Illegal Quantity Error" occurs. Do not confuse SPC with SPACE$. The SPC function does not return a string value like the SPACE$ function, but merely instructs the PRINT statement to skip a certain number of spaces. If you attempt to use the SPC function in any statement other than PRINT a syntax error will occur.

Note that if the SPC function appears at the end of a PRINT argument list with or without a following semicolon, a carriage return is not output.

See Also

PRINT
PRINT#
TAB

Example

```
PRINT "xxx"; SPC(3)
PRINT "yyy"; SPC(3)
PRINT "zzz"

myString$ = SPC(10)     'This is an ILLEGAL use of the SPC function
```

OUTPUT:

```
xxx   yyy   zzz
```

SQR Function

Syntax

```
    SQR(NumericExpression)
```

Action

The SQR function returns the square root of *NumericExpression*.

NumericExpression must be a positive number, otherwise a runtime error occurs. The SQR function is faster than raising a number to the 0.5 power.

Example

```
FOR i = 1 to 10
   PRINT i, SQR(i)
NEXT i
```

OUTPUT:

```
1           1
2           1.414214
3           1.732051
4           2
5           2.236068
6           2.44949
7           2.645751
8           2.828487
9           3
10          3.162278
```

STOP Statement

Syntax

```
    STOP
```

Action

The STOP statement aborts the execution of the program, closes all open files and causes the runtime error: "Program Interrupted".

Example

```
STOP
```

STR$ Function

Syntax

```
    STR$(NumericExpression)
```

Action

The STR$ function evaluates the given *NumericExpression* and returns the value as a string. That is, it returns a string which is equivalent to what you would see on the screen if you were to PRINT *NumericExpression*.

The complementary function is VAL, which takes a string argument and returns a numeric value.

See Also

VAL

Example

```
someNum = 123.456           'Set the variable someNum

someString$ = STR$(someNum)  'Convert the value to a string

PRINT  someNum, someString$  'Make sure they print the same thing

IF someNum = VAL(someString$) THEN
   PRINT "STR$ and VAL work!"
END IF
```

OUTPUT:

```
123.456     123.456
STR$ and VAL work!
```

SUB$ Statement

Syntax

```
    SUB$(StringVariable, Start [,Count] ) = StringExpression
```

Action

The SUB$ statement replaces a substring of a string variable with another string value.

SUB$ substitutes the value of *StringExpression* into the *StringVariable* beginning with the *Start* character in the string. If the optional *Count* parameter does not appear, then the number of substituted characters is the number of characters in *StringExpression,* otherwise only *Count* characters are replaced. If *Start* is greater than the number of characters in the *StringVariable,* then the SUB$ statement does nothing. Both *Start* and *Count* are integer expressions which should be in the range 1 through 255.

Example

```
Str$ = "TML BASIC "
SUB$(Str$,5) = "Pascal"
PRINT Str$

Str$ = "TML BASIC "
SUB$(Str$,5,2) = "Pascal"
PRINT Str$
```

OUTPUT:

```
TML Pascal
TML PaSIC
```

SWAP Statement

Syntax

```
    SWAP(Variable1,Variable2)
```

Action

The SWAP statement exchanges the values of two variables.

Variable1 and *Variable2* are two simple variables or array elements of the same type. If the two variables are not of the same exact type, a "Type Mismatch Error" occurs.

The SWAP statement is handy because it is not possible to simply trade the values of two variables with the statements:

    var1 = var2 : var2 = var1

When the second assignment is done, the value of *var1* already has the value of *var2*. Instead, it is necessary to introduce a temporary variable and a third statement:

    temp = var1 : var1 = var2 : var2 = temp

Example

```
var1% = 10
var2% = 43
PRINT var1%, var2%

SWAP var1%, var2%
PRINT var1%, var2%
```

OUTPUT:

```
10            43
43            10
```

TAB Function

Syntax

`TAB(NumericExpression)`

Action

The TAB function is used to tab to the specified print position *NumericExpression* in a PRINT or PRINT# statement.

The TAB function causes the current print position to move to the *NumericExpression* space from the left margin of the line. If the current print position is already beyond the specified position, the TAB function has no effect.

NumericExpression must be an integer value in the range 1 through 255, otherwise an "Illegal Quantity Error" occurs. If you attempt to use the TAB function in a statement other than PRINT a syntax error will occur.

Note that if the TAB function appears at the end of a PRINT argument list with or without a following semicolon, a carriage return is not output.

See Also

PRINT
PRINT#
SPC

Example

```
FOR i% = 1 TO 10
   PRINT "X"; TAB(i%); "Y"
NEXT i%
```

OUTPUT:

```
XY
XY
X Y
X  Y
X   Y
X    Y
X     Y
X      Y
X       Y
X        Y
```

TAN Function

Syntax

```
    TAN(NumericExpression)
```

Action

Returns the trigonometric tangent of *NumericExpression*. *NumericExpression* is an angle expressed in radians. To convert radians to degrees, multiply by 180/π. To convert degrees to radians, multiply by π/180.

See Also

ATN
COS
PI
SIN

Example

```
PRINT "Tangent of 45 degrees = '; TAN(45 * PI/180)
```

OUTPUT:

```
1
```

TEN Function

Syntax

```
    TEN(HexStringExpression)
```

Action

The TEN function returns the decimal (base 10) equivalent of the hex digits in the specified *StringExpression*. The returned value is a double integer. *HexStringExpression* may contain leading spaces followed by an optional dollar sign character, but the next eight or fewer characters of the string must represent a hexadecimal number; otherwise an "Illegal Quantity Error" occurs.

See Also

HEX$

Example

```
PRINT TEN("$E1000")
```

OUTPUT:

```
921600
```

TASKPOLL INIT Statement
TASKPOLL Statement

Syntax

```
TASKPOLL INIT NumericExpression
TASKPOLL NumericExpression
```

Action

The TASKPOLL statements are used for writing event-driven, desktop programs. The TASKPOLL INIT statement is used to define the types of events detected, while the TASKPOLL statement is used to actually detect events. Writing event-driven programs is not a trivial task, be sure to have a solid understanding of the information presented in Chapters 11 through 13 before writing event-driven programs.

The TASKPOLL statements in TML BASIC use the Toolbox Window Manager *TaskMaster* routine for detecting events. Thus, to use the TASKPOLL statements in a program it is necessary for a program to properly load and initialize the required desktop tools, in particular the Window Manager. For complete information on what tools are required, and how to load and initialize them see Chapter 13.

As discussed with the EVENTDEF statement, the TASKPOLL statement is capable of detecting 29 different events. The first 16 events are the standard Event Manager event types, while the remaining 13 events are those detected by TaskMaster when a mouse-down event occurs in special places on the desktop. It is possible to control which of the latter 13 events are actually detected by TaskMaster by setting its *TaskMask*. TaskMask is an integer value specified by the *NumericExpression* argument to the TASKPOLL INIT statement. Following are the individual TaskMask values. The individual values are added together to form the complete TaskMask. Thus, a TaskMask value of 8191 (sum of all values) would indicate that TaskMaster should detect every possible event type.

1	Detect menu keys
2	Perform automatic window updating
4	Perform FindWindow
8	Perform MenuSelect
16	Perform OpenNDA
32	Perform SystemClick
64	Perform DragWindow
128	Perform SelectWindow if mouse down in content
256	Perform TrackGoAway
512	Perform TrackZoom
1024	Perform GrowWindow
2048	Perform automatic scrolling support
4096	Handle special menu items

The TASKPOLL INIT statement must be executed before the TASKPOLL statement in order to define a TaskMask value. If a program must change the TaskMask, the TASKPOLL INIT statement can be executed again.

The TASKPOLL statement is used to detect events. When an event occurs, the TASKPOLL statement examines the *Event Dispatch Table* and *Menu item Dispatch Table* for the appropriate event-handling subroutine. These tables are defined by the EVENTDEF and MENUDEF statements. Normally, a program executes the TASKPOLL statement in a loop, allowing the TASKPOLL statement to automatically call event-handling subroutines.

The *NumericExpression* argument defines the *EventMask*. The EventMask is used to indicate which events should be returned by the TASKPOLL statement. If an event exists, but the EventMask indicates the event should not be returned, it remains in the Event Manager event queue until TASKPOLL is called with an EventMask which allows the event to be returned.

The following are the individual EventMask values. The individual values are added together to form the complete EventMask. Thus, a EventMask value of -1 (all bits set) indicates that TaskMaster should return every possible event type.

2	Mouse down
4	Mouse up
8	Key down
32	Auto key
64	Update
256	Activate
512	Switch
1024	Desk Accessory
2048	Device Driver
4096	Application defined #1
8192	Application defined #2
16384	Application defined #3
-32768	Application defined #4

See Chapter 13 for a complete discussion of how to write event-driven programs.

See Also

EVENTDEF
EXEVENT@
MENUDEF
TASKREC%
Chapters 11 through 13

TASKREC% and TASKREC@ Functions

Syntax

```
    TASKREC%(NumericExpression)
    TASKREC@(NumericExpression)
```

Action

The TASKREC functions return a single or double integer result from the Task Master TaskRec. *NumericExpression* is an integer value which represents a word offset into the *Task Record data structure.* A Task Record is an internal TML BASIC variable which is declared as an Event Manager Event Record. The definition of an Event Record is as follows (from Appendix C):

```
    DIM anEventRecord!(19)
```

Element(s)	Value	Description
0..1	Integer	Event code specifying which event occurred
2..5	Double Integer	Event message which has additional information about event
6..9	Double Integer	Number of ticks since startup
10..13	*Point*	Mouse location where event occurred
14..15	Integer	Modifier flags
16..17	Double Integer	Task Data for Task Master
18..19	Double Integer	Task Mask for Task Master

For example, the function call *TASKREC@(2)* returns the Event Message field from the Event Record as a double integer value.

The TASKREC functions are used in event-handling subroutines. An event-handling subroutine extracts information in the event record to determine exactly what action to take in response to an event. For example, the TASKREC function can be used to determine the location of the mouse for a mouse-down event.

The meaning of each field depends upon the event type returned. For a complete description of meaning for these fields see Chapter 13. The *Apple IIGS Toolbox Reference* is also a good source of information regarding this data structure.

TEXT Statement

Syntax

```
    TEXT
```

Action

The TEXT statement sets the display screen to the full screen text mode, clearing any other text or graphics mode in use. The cursor is positioned on the first line at the left margin at the top of the screen.

See Also

GRAF INIT
GRAF ON
GRAF OFF
HOME

Example

```
TEXT
```

TEXTPORT Statement

Syntax

```
    TEXTPORT Left,Top TO Right, Bottom
```

Action

The TEXTPORT statement sets the size and position of the text window within the text screen. After executing the TEXTPORT statement, all subsequent output to the text screen is constrained to the text window. The remainder of the text screen is undisturbed.

Left, Top, Right, Bottom define the boundries of the text window. If the specified text window boundries define a window larger than the size of the text screen (24 rows by 80 columns) then the text window is truncated to fit.

Example

```
TEXTPORT 10,10  TO 20,20
```

TIME Function

Syntax

```
    TIME(NumericExpression)
```

Action

The TIME function reads the Apple IIGS clock to return the current time information as an integer rather than a string, as returned by the TIME$ function. The value of *NumericExpression* must be in the range 0 through 3 inclusive, otherwise the "Illegal Quantity Error" occurs.

The following table shows the values returned by the TIME function for each legal parameter value.

Function	**Value returned**
TIME(0)	Hour (0 through 23), reads the clock
TIME (1)	Hour (0 through 23)
TIME (2)	Minute (0 through 59)
TIME (3)	Second (0 through 59)

Actually, the Apple IIGS clock is only read when the parameter value is zero. TIME(0) reads the Apple IIGS clock for all time information and then updates the values which will be returned by the other TIME function calls. This feature protects programs from classical "clock rollover" problem.

Example

```
ReadTime%   = TIME(0)    'Read the Apple IIGS Time information
Hour%       = TIME(1)
Minute%     = TIME(2)
Second%     = TIME(3)

PRINT "The time is "; Hour%; ":"; Minute%; ":"; Second%
```

OUTPUT:

```
The time is 14:32:15
```

TIME$ Statement

Syntax

```
    TIME$
    TIME$ Hour, Minute, Second
```

Action

TIME$ is both a function and a statement in TML BASIC. The TIME$ statement has three arguments, while the TIME$ function has none.

The TIME$ function reads the Apple IIGS clock and returns the current time as a string. The form of the string depends upon the time format chosen in the Apple IIGS control panel. The time format is HH:MM:SS, where HH stands for the hour, MM stands for the minute and SS stands for the second. The variations of this format are between a 12 and 24 hour clock. See your *Apple IIGS Owner's Guide* for information on how to use the Control Panel.

The TIME$ statement is used to change the time settings of the Apple IIGS clock. The hour is specified by the *Hour* parameter, the minute by the *Minute* parameter and the second by the *Second* parameter. The *Hour* parameter should be in the range 0 through 23, while the *Minute* and *Second* parameters should be in the range 0 through 59.

Example

```
TIME$ 8, 20, 40       'Set the clock to 8:20:40 AM.
PRINT TIME$
```

OUTPUT:

```
 8:20:40 AM
```

TIMER ON and TIMER OFF Statement

Syntax

```
    TIMER ON
    TIMER OFF
```

Action

The TIMER ON statement initiates a process which reads the time from the Apple IIGS clock and calculates a "seconds from midnight" (a number from 0 to 86399) value and stores it in the SECONDS@ counter variable. The counter is updated once a second using the Apple IIGS 1-second clock interrupt. After TIMER ON is executed, a program may initiate a one-shot interval timer using the ON TIMER statement. TIMER OFF disables the 1-second clock interrupt and thus freezes the SECONDS@ counter.

Due to the low priority of the 1-second interrupt and other factors in the system, it is possible for the counter to miss an interrupt and not reflect the actual number of seconds since midnight.

See Also

ON TIMER
SECONDS@

Example

```
TIMER ON
RANDOMIZE SECONDS@
```

TYP Function

Syntax

```
    TYP(FileNumber)
```

Action

The TYP function returns the type of the next data item to be read from the specified Basic Data File on its next access. The function is typically used to ensure that a READ statement with a variable of the proper type is used when the exact elements of the file are not known.

The *FileNumber* argument is the file reference number of the file previously opened with that number. The number returned by the TYP function denotes what type of data will next be read from the specified file. TYP returns the following values:

Value	Meaning
0	End of File
2	Next datum is of type Integer (%)
3	Next datum is of type Double Integer (@)
4	Next datum is of type Long Integer (&)
5	Next datum is of type Single Real (no suffix)
6	Next datum is of type Double Real (#)
7	Next datum is of type String ($)

If the type of file referenced by *FileNumber* is not a Basic Data File (BDF), a "File Type Error" occurs. If *FileNumber* does not refer to a currently open file, the "File Not Open Error" occurs.

Example

```
nextTyp% = TYP(1)
IF nextTyp% = 2 THEN
   READ# 1; nextInt%
ELSE IF nextTyp% = 3 THEN
   READ# 1; nextDblInt@
ELSE IF nextTyp% = 4 THEN
   READ# 1; nextLongInt&
ELSE IF nextTyp% = 5 THEN
   READ# 1; nextSglReal
ELSE IF nextTyp% = 6 THEN
   READ# 1; nextDblReal#
ELSE IF nextTyp% = 7 THEN
   READ# 1; nextString$
END IF
```

UBOUND Function

Syntax

```
    UBOUND(ArrayName [()] [,DimNumber] )
```

Action

The UBOUND function returns the upper bound (largest possible subscript) of the specified dimension of an array. The array is given by *ArrayName,* optionally followed by the left and right parentheses. If the array is a multi-dimensional array, the optional *DimNumber* can be used to specify which dimension UBOUND should return as the upper bound. If *DimNumber* is not specified then UBOUND returns the upper bound of the first dimension of the array.

UBOUND is typically used with dynamic arrays in order to determine their current size. The lower bound of an array is always zero.

See Also

DIM

Example

```
i% = 5
j% = 45
DIM DYNAMIC  someArray(i%,j%)
PRINT UBOUND(someArray,1), UBOUND(someArray,2)
```

OUTPUT:

```
5             45
```

UCASE$ Function

Syntax

```
    UCASE$(StringExpression)
```

Action

The UCASE$ function returns a string which is the value of *StringExpression* with all lowercase letters, a through z, converted to upper case letters, A through Z.

Example

```
PRINT UCASE$("tml BaSic")
```

OUTPUT:

```
TML BASIC
```

VAL Function

Syntax

```
VAL(StringExpression)
```

Action

The VAL function evaluates *StringExpression* and returns the value as a real or integer number.

The evaluation is carried out from left to right. If the beginning characters of *StringExpression* do not evaluate to a legal numeric value, the value returned is zero. If *StringExpression* begins with legal numeric characters followed by non-numeric characters, only the numeric characters are evaluated.

The complementary function is STR$, which takes a numeric argument and returns a string value.

See Also

STR$

Example

```
someString$ = "123.456"       'Set the variable someString$

someNum = VAL(someString$)    'Convert the value to a number

PRINT someString$, someNum    'Make sure they print the same thing

IF someString$ = STR$(someNUM) THEN
   PRINT "VAL and STR$ work!"
END IF
```

OUTPUT:

```
123.456     123.456
VAL and STR$ work!
```

VAR Function

Syntax

```
VAR(StructureArrayReference, VariableType [,Length])
VAR(MemoryAddress, VariableType [,Length])
```

Action

The VAR function is used to extract values from a structure array variable. This statement is the inverse of the SET statement.

The `StructureArrayReference` must be a structure array variable which has been previously declared. The *VariableType* argument is used to indicate the type of the value to be extracted from the structure array. The type implicitly defines the number of bytes to be extracted from the array. Following are the legal values for the *VariableType* argument:

1	extended real
2	integer
3	double integer
4	long integer
5	single real
6	double real
7	string

The above values are the same as the values returned by the TYP function. The only exception is the value 1, since extended reals are not stored in BDF files.

The *Length* parameter may be used with the integer types to specifiy a size smaller than the default integer sizes (2, 4, 8), and it must be used with the string type. For integers, *Length* may be 1 or 2; for double integers, *Length* may be 1, 2, 3 or 4; and for long integers, it may be 1 through 8. When an integer is created from a reduced size, the result is always a positive number; that is, no sign extension is provided. For strings, *Length* must be in the range from 1 to 255.

The second form of the VAR function effectively implements a multibyte peek operation. `MemoryAddress` is a double integer which specifies a location in memory that is essentially treated as a structure array. The *VariableType* and *Length* parameters are used as above to control the data which is extracted.

See Also

SET
TYP

Example

```
i% = VAR(aStruct!(0),2)          'Extract two bytes as an integer
i% = VAR(aStruct!(0),2,1)        'Extract one byte as an integer

msg$ = VAR(aStruct!(12),7,10) 'Extract 10 bytes as a string

i% = VAR(aPointer@,2)            'A multi-byte peek to read an integer
```

VAR$ Function

Syntax

```
    VAR$(MemoryAddress [,Length])
```

Action

The VAR$ function creates a TML BASIC string value from the counted string at the memory address specified by *MemoryAddress*. *MemoryAddress* is a numeric expression which must be a legal Apple IIGS address pointing to a counted string. The optional *Length* parameter specifies the number of characters to extract from memory.

The VAR$ function is typically used to extract a counted string from a data structure returned by an Apple IIGS Toolbox routine.

VARPTR and VARPTR$ Functions

Syntax

```
    VARPTR(VariableName)
    VARPTR$(StringVariable)
```

Action

The VARPTR function returns the address of the indicated *VariableName*. For string variables, VARPTR will return the address of the string descriptor, NOT the address of the string data. To obtain the address of an array, specify the array name indexed by zero.

The VARPTR$ function is used to obtain the address of the string data for a string variable. If the specified string variable is a null string, VARPTR$ returns zero. If a numeric variable is passed to VARPTR$, a "Type Mismatch Error" occurs.

Both VARPTR and VARPTR$ return a double integer. If an undefined variable is specified as the argument to these functions, the "Undefined Variable" error is reported.

Example

```
DIM myArray%(50)

theAddr@ = VARPTR(myArray%(0))     'Address of the array myArray%
theAddr@ = VARPTR(theAddr)         'Address of the simple variable theAddr
theAddr@ = VARPTR(myString$)       'Address of the string variable myString$
theAddr@ = VARPTR$(myString$)      'Address of the string variable's data
```

VOLUMES Statement

Syntax

```
VOLUMES
```

Action

The VOLUMES statement is used to read the volume name for each ProDOS 16 device and display its name. The ProDOS 16 devices are numbered .D1 through .D9 inclusive. The display lists the device name, its volume name and the number of free bytes of storage available on the volume.

See Also

Chapter 9

WRITE# Statement

Syntax

```
    WRITE# FileNumber [, RecordNumber ] [; Expression {,|; Expression }]
```

Action

The WRITE# statement writes information to a BDF file. The reserved word WRITE# is followed by the file reference number of an open file to write to, a semicolon, and a list of expressions separated by commas or semicolons.

This form of the WRITE# statement performs sequential access, writing each successive value at the *current file position.* Each expression in the WRITE# argument list causes a *field* to be written to the BDF file. Recall that a field is a tag byte followed by the binary representation of the value. If a record does not contain enough room to hold all the fields being written to it, the extra fields are written to the next record. If a field cannot fit in any record (it is larger than the record size), an error occurs.

An optional form of the WRITE# statement permits random access to a BDF file. To perform random access using the WRITE# statement, include a record number after the file reference number.

See Also

READ#
Chapter 9

Example

```
WRITE #10; anInt%, aReal, aStr$
WRITE #10; "hello"
WRITE #10,6; anInt%, aReal, aStr$
```

Part III

Toolbox Programming

Chapter 11
Programming the Toolbox

In this and the following two chapters, the techniques for writing TML BASIC programs which utilize the features of the Apple IIGS Toolbox are discussed. The Toolbox is the name used to designate the large collection of software routines developed by Apple Computer and built into every Apple IIGS. The Toolbox routines implement features like drawing to the Super Hi-Res graphics screen, sound, menus, windows, dialogs, and much more. Each of the Toolbox routines are available in TML BASIC for creating programs which make use of these advanced capabilities of the Apple IIGS.

The Toolbox software is organized into several functional components called *tool sets* (or *managers*). Each tool set defines a collection of procedures, functions and data structures. For example, the tool set responsible for the creation and manipulation of windows on the screen is called the *Window Manager,* the tool set responsible for drawing in the Super Hi-Res graphics screen is called *QuickDraw,* and so on. In addition to providing applications with a powerful collection of software, the Toolbox also serves to insulate your program from the details of machine hardware. By using the tool sets, Apple Computer can actually change the Apple IIGS hardware, provide new versions of the tool sets without affecting your program's ability to operate.

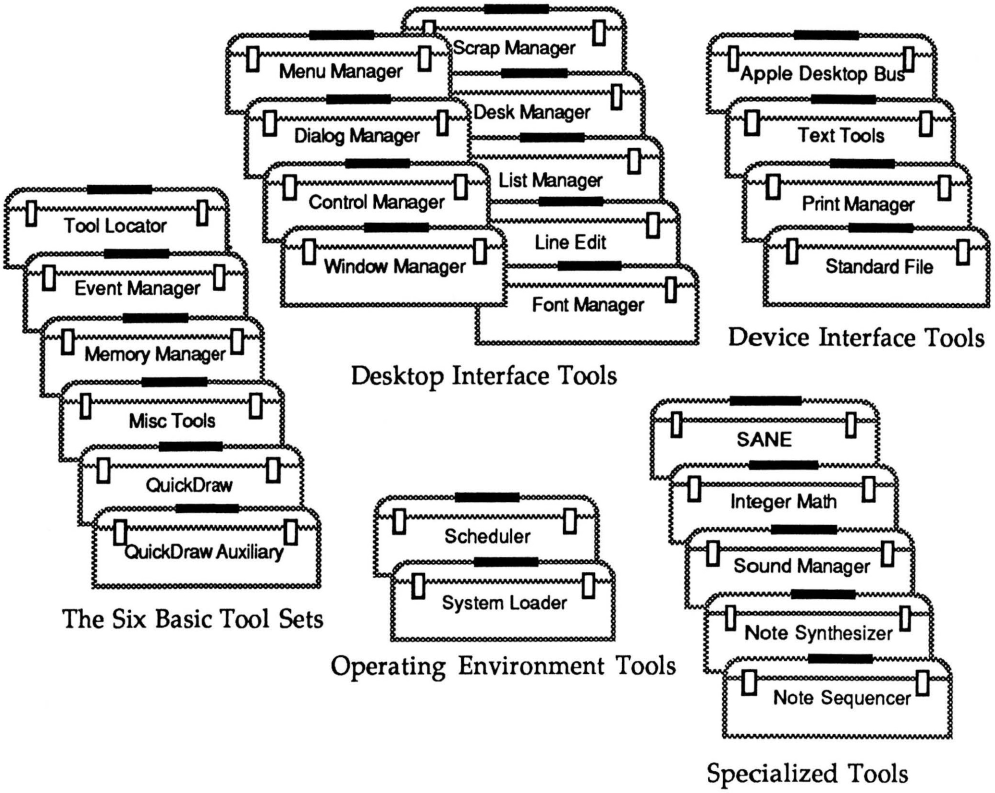
This chapter provides a review of the Apple IIGS Toolbox and its organization. Also discussed are the features available in the TML BASIC language for programming with the Toolbox. Chapter 12 provides a thorough discussion of the QuickDraw graphics engine. Together, the Super Hi-Res Graphics screen and the QuickDraw graphics engine form the foundation of the Apple IIGS's graphics capabilities. Most all of the other tool sets rely upon QuickDraw for their implementation, thus a good understanding of how QuickDraw works is necessary to your success in programming the other tool sets. Finally, Chapter 13 discusses the issues and TML BASIC features related to creating *event-driven* programs which use the *Apple Desktop Interface.* Programs which use the Apple Desktop Interface are those which make use of the mouse, menus, windows, dialogs, etc.

Appendix C provides a complete reference for the TML BASIC predefined Toolbox libraries. The appendix contains a listing of every procedure and function in the tool sets.

Review of the Apple IIGS Toolbox

The following paragraphs are an introduction to the various tool sets in the Apple IIGS Toolbox. While it is intended this information provide you with a good understanding of the contents and organization of the Toolbox, it is by no means complete. The *Apple IIGS Toolbox Reference, Volumes 1 and 2,* published by Apple Computer, is the most complete and thorough documentation on the Toolbox and is absolutely necessary for writing programs which make significant use of the Toolbox.

The Apple IIGS Toolbox consists of numerous tool sets implementing a broad range of operations. The tool sets may be grouped into five major functional categories: the six basic tools, the desktop interface tools, the device interface tools, the operating environment tools and the specialized tools. The following diagram illustrates the functional organization of the tool sets.



The Six Basic Tool Sets

The six basic tool sets provide the framework upon which all of the other tools are built upon. All of these tools are used in event-driven programs.

Tool Locator	The Tool Locator is the most important of the Apple IIGS tool sets. The Tool Locator allows you to load tool sets from disk into RAM and is responsible for locating a tool set routine when a program calls a Toolbox procedure or function.
Memory Manager	The Memory Manager is the second most important tool set. This tool is entirely responsible for the allocation, deallocation, and repositioning of memory blocks on the Apple IIGS. The Memory Manager keeps track of how much memory is free and what parts are allocated and to whom. Whenever a program needs memory, it must ask the Memory Manager to allocate it.
Miscellaneous Tools	The Miscellaneous Tools consist mostly of system-level routines that must be available to most other tool sets.
QuickDraw	QuickDraw is the tool set that controls the graphics environment of the Apple IIGS and draws simple objects and text in the Super Hi-Res graphics screen. All other tools which create graphical objects such as the Menu and Window Manager call the QuickDraw tool set.
QuickDraw Auxiliary	This tool contains additional graphics routines which complement the QuickDraw tool set.
Event Manager	The Event Manager is responsible for detecting system events such as mouse-clicks, keystrokes, window updates, etc. It queues the events and then delivers the events to an application as requested.

Desktop Interface Tools

The tool sets in this group support the Apple Desktop Interface. The desktop interface is the visual interface between the user of an application and the computer. It includes the menu bar and the blue colored area on the screen. Applications usually have documents on the desktop displayed in windows and perhaps other graphic objects such as icons. Applications implementing the desktop will always use the Menu, Window and Control managers, and usually most of the others as well. New Desk Accessories are supported by the Desk Manager.

Control Manager	The Control Manager consists of all the routines necessary to manipulate controls. Examples of controls include scroll bars, radio buttons, check boxes, etc.
Desk Manager	The Desk Manager is the tool which enables an application to support both classic desk accessories and new desk accessories.
Dialog Manager	The Dialog Manager provides the routines which allow an application to create and use both dialog boxes and alerts as a means of communication between a user and your program.
Font Manager	The Font Manager is the tool set which allows an application to make use of different text fonts, font styles, etc. within QuickDraw.
Line Edit	Line Edit is used to display and edit a line of text on the screen and allow a user to edit the text.
List Manager	The List Manager is used to create, display and allow selection of a variable amount of similar data.
Menu Manager	The Menu Manager controls and maintains the use of *pull-down menus* and items in the menus.
Scrap Manager	The Scrap Manager implements the *desk scrap,* which implements the *Cut, Copy,* and *Paste* operations of an application.
Window Manager	The Window Manager creates the desktop environment and is responsible for the creation and manipulation of windows.

Device Interface Tools

The tool sets in this group are used to manage input and output between the computer and peripheral devices and a program.

Apple Desktop Bus	The Apple Desktop Bus is a method and a protocol for connecting input devices, such as keyboards and mice with the Apple IIGS. The routines in this tool set are used to send commands and data between the Apple Desktop Bus Microcontroller and the rest of the system.
Print Manager	The Print Manager allows an application to use QuickDraw routines to print text and graphics to an Imagewriter or LaserWriter.
Standard File	The Standard File tool set implements the standard user interface for specifying a file to be opened or saved.
Text Tools	The Text Tools provide an interface between the Apple II character device drivers, which must be executed in emulation mode, and applications running in native mode.

Operating Environment Tools

The operating environment tools control the interaction between low-level hardware and software functions. While not listed here, the Memory Manager and Miscellaneous Tools tool sets implement similar low-level operations characteristic of the Operating Environment tools and in many cases interact with these tool sets.

Scheduler	The Scheduler delays the activation of a desk accessory or other system task until the resources that the task/desk accessory requires become available. This avoids potential system crashes when more than one task attempts to use the same resource at the same time.
System Loader	The System Loader is responsible for loading and relocating code for applications and desk accessories to memory.

Specialized Tools

The specialized tool sets are those which do not fit into any of the above groups. These tool sets can be grouped into two categories: those related to sound and those related to mathematical operations. The sound tools provide access to the powerful sound capabilities of the Apple IIGS sound hardware, in particular the ENSONIQ DOC chip. The mathematical tools implement floating-point and integer calculations.

Sound Manager The Sound Manager provides access to the Apple IIGS's sound hardware for creating basic sounds.

Note Synthesizer The Note Synthesizer is used to create complex musical sounds simulating a variety of instruments using the Apple IIGS's sound hardware.

Note Sequencer The Note Sequencer is used to string together notes from the Note Synthesizer into sequences, patterns and phrases that make up a song.

Integer Math This tool set consists of a varied collection of operations for integers, long integers and signed fractional numbers. These include multiplication, division, conversions, etc.

SANE SANE implements the Standard Apple Numeric Environment. It is an extended-precision IEEE 754 conformant implementation of floating point arithmetic and transcendental functions.

Where are the Tools?

As of System Disk release version 3.1, 28 tool sets have been defined for the Apple IIGS Toolbox. Many of these tools are located in the Read-Only Memory (ROM) built into every Apple IIGS. Other tool sets are located on disk and must be loaded into Random Access Memory (RAM) before they are used. Some of the RAM based tool sets are located in RAM because there was not sufficient space left in the ROM to include the tool set there, while others are in RAM because they were either not completed or have been added since the ROM was created. Fortunately, a program does not have to concern itself with the exact location of a particular procedure or function in a tool set. This is the responsibility of the Tool Locator tool set. This design for the Toolbox allows Apple Computer to move existing tool sets into ROM, move the location of tool sets within ROM or RAM and even add new tool sets to the Toolbox without affecting existing programs.

The only requirement of a program using the Toolbox is to ensure the tool sets used by a program which are not in ROM are available on the system disk so that they can be loaded into RAM. Tool sets residing on disk must be located in the TOOLS folder within the SYSTEM folder of the system disk used to *boot* the Apple IIGS. The name of a tool set file is TOOLxxx where the xxx is a three digit number corresponding to the tool set's assigned tool number. For example, the Window Manager tool set file has the name TOOL014 because the Window Manager is tool set number 14. The TML BASIC distribution disk is shipped with all of the RAM based tool sets available on the Apple IIGS System Disk.

Table 11-1 lists of all 28 Apple IIGS tool sets with their names and whether or not the tool set is currently located in ROM or RAM. This information is accurate for

System Disk version 3.1 and ROM version 01.

Table 11-1
Apple IIGS Toolbox

Tool Number	Tool Name	RAM	ROM
1	Tool Locator	-	X
2	Memory Manager	-	X
3	Miscellaneous Tools	-	X
4	QuickDraw II	-	X
5	Desk Manager	-	X
6	Event Manager	-	X
7	Scheduler	-	X
8	Sound Manager	-	X
9	Apple Desktop Bus	-	X
10	SANE	-	X
11	Integer Math	-	X
12	Text Tools	-	X
13	*Reserved for System Use*	-	-
14	Window Manager	X	-
15	Menu Manager	X	-
16	Control Manager	X	-
17	System Loader	X	-
18	QuickDraw Auxilary Routines	X	-
19	Print Manager	X	-
20	Line Edit	X	-
21	Dialog Manager	X	-
22	Scrap Manager	X	-
23	Standard File	X	-
24	Disk Utilities	X	-
25	Note Synthesizer	X	-
26	Note Sequencer	X	-
27	Font Manager	X	-
28	List Manager	X	-

The Toolbox Libraries

Recall from Chapter 8 that a *library* is an independent collection of TML BASIC source code which is compiled separately from a program. A compiled library can be used in other libraries and programs. In addition to allowing user defined libraries, TML BASIC provides several predefined libraries which define the interfaces to each Apple IIGS Toolbox tool set. These predefined libraries are called the *Toolbox Libraries*. The toolbox libraries are found in the LIBRARIES folder on the TML BASIC distribution disk.

Each of the Toolbox Libraries contain procedure and function declarations for a particular tool set. The source code to these libraries is not shipped with TML BASIC, however, Appendix C provides a complete listing of the source code for these libraries. For example, the QuickDraw library contains declarations for every procedure and function defined for the QuickDraw tool set. Note, the code implementing each procedure and function is not given after the DEF PROC and DEF FN statements. Instead, the special TOOL directive is specified after the parameter list. The TOOL directive is followed by two integers: the *function number* and *tool set number* respectively. TML BASIC uses this information to tell the Tool Locator how to locate the code for a toolbox procedure or function. Note that the TOOL directive is not legal TML BASIC, and thus, cannot be used in your own programs. The TOOL directive is a special extension only used to define the toolbox libraries.

Each of the tool sets in the Toolbox is assigned a unique number called the *tool set number*. Given this number, the Tool Locator knows which tool set a toolbox procedure or function belongs. In addition, each procedure and function within a given tool set is assigned a unique integer called the *function number*. The tool set number and function number together are used by the Tool Locator to uniquely identify every procedure and function in the Toolbox. For example, the *MoveTo* procedure in the QuickDraw tool set is declared:

```
DEF PROC MoveTo(H%,V%)  TOOL 58,4
```

This declaration defines the *MoveTo* procedure to have two integer parameters (representing the horizontal and vertical position to locate the QuickDraw pen), and that the procedure is implemented as function number 58 in the QuickDraw tool set (tool set number 4).

The LIBRARY Statement

To use a tool set library in a program its name must be specified in a LIBRARY statement. When the LIBRARY statement is used, TML BASIC enters all of the procedure and function declarations of the library into its symbol table just as if the declarations had been made in the source code. For example, the QuickDraw library can be used in a program with the following statement:

```
LIBRARY "QuickDraw"
```

The LIBRARY statement can appear anywhere in a program. Before TML BASIC compiles a program, it first scans the file for all occurrences of the LIBRARY statement. As each LIBRARY statement is encountered, its declarations are entered into the program's symbol table, making them available throughout the entire program.

The LIBRARY statement also serves another purpose. As noted above, several of the tool sets are not available in ROM, but rather are implemented in disk files which must be loaded into RAM. When a LIBRARY statement names a tool set which is not in ROM, TML BASIC automatically generates code to load the disk file into RAM. To do this, it generates a toolbox call to the LoadOneTool procedure in the Tool Locator tool set. See Chapter 13 for more information about loading tool sets into RAM.

Searching for a Library

When a library name is specified in the LIBRARY statement, TML BASIC searches for the library's compiled *library file*. The library file is not the source code for the library, but its compiled declarations and code. As described in Chapter 3, the name for a library file is the name of the library with the suffix ".LIB". For example, the library filename for the toolbox library QuickDraw is QUICKDRAW.LIB.

TML BASIC searches in three locations to find a library file. First, it looks to see if the library file is already in memory. Second, it searches in the same folder as the source code file containing the LIBRARY statement. And finally, if the file is not found there, it searches in the directory specified in the *Library Search Path* option of the Preferences Dialog (see Chapter 6 for more information about the Preferences Dialog). If the file is not found in any of these locations, TML BASIC then reports an error.

It is possible to override TML BASIC by specifying the complete pathname of the library file. For example, the following statement indicates the QuickDraw library file is in the /TML/LIBRARIES subdirectory:

```
    LIBRARY "/TML/LIBRARIES/QUICKDRAW"
```

Note, even though the complete pathname is specified, the ".LIB" suffix is not included. This is because TML BASIC automatically adds the suffix regardless of whether the full pathname is used or not.

The CALL Statement

While the toolbox procedures and functions are declared using the DEF PROC and DEF FN statements as shown in Appendix C, they are not normal procedures and functions. As such, they are not called using the PROC and FN reserved words. Instead they are called using the CALL statement. The CALL statement is a special reserved word used to indicate that a program is calling a Toolbox procedure or function. For example, the QuickDraw *MoveTo* procedure is called as follows:

```
    CALL MoveTo(10,23)
```

TML BASIC allows the use of the underscore symbol (_) as a shorthand form of the CALL reserved word. Thus, the statement shown above can be rewritten as follows:

```
_MoveTo(10,23)
```

If a program attempts to CALL a toolbox procedure or function in a tool set which has not been named in a LIBRARY statement within the program, TML BASIC will report the error "Toolbox procedure *xxx* is not defined", where *xxx* is the name of the procedure or function.

When calling a toolbox procedure, the number and type of parameters must match the declaration of the procedure in its library file. If the parameter list does not match, TML BASIC reports an error. The rules for matching parameters are the same as for normal BASIC procedures and functions.

As described in Chapter 7, TML BASIC stores strings in a data structure called the *string pool,* and the value of a string variable is an integer offset into the string pool where the string data is stored. Whenever a toolbox procedure has a string parameter, TML BASIC automatically converts the value of a string variable into the machine address of the string data and passes that value for the parameter. Toolbox procedures with string parameters expect the address of the string data, where the string data is stored as a *counted string.* A counted string is represented as an integer byte whose value is the number of characters in the string followed by the actual characters in the string. Unlike GS BASIC, TML BASIC stores string data in the string pool as counted strings, therefore, no conversion is necessary.

The R.STACK Functions

Many of the toolbox routines are implemented as functions, which of course return values. In addition, every toolbox routine returns an error code indicating whether the routine executed successfully or not. The error code and function result values are saved in a special data structure called the *CALL return stack.* The stack is a 32 byte buffer (16 words). The R.STACK functions are used to read values out of the stack. The syntax for calling the R.STACK function is as follows:

```
R.STACK%(NumericExpression)
R.STACK@(NumericExpression)
R.STACK&(NumericExpression)
```

Because each toolbox function can return a different amount of information and different data types, the CALL return stack can be accessed for integer, double integer and long integer values. The *NumericExpression* parameter is a *word* offset into the stack. The number of bytes read from the stack beginning at that point depends upon which R.STACK function is called.

R.STACK% returns an integer value reading 2 bytes of data from the stack; the R.STACK@ returns a double integer reading 4 bytes of data from the stack; and finally, R.STACK& returns a long integer reading 8 bytes of data from the stack. Thus, R.STACK% may be indexed in the range 0 to 16, R.STACK@ in the range 0 to 15, and R.STACK& in the range 0 to 13.

R.STACK%(0) returns the *error code* returned by the toolbox procedure or function. If the value is zero, then no error occurred. If the value is non-zero, an error occurred during the execution of the toolbox routine, and your program should take appropriate action.

R.STACK%(1) is the first word of data returned on the CALL stack.

The following code fragment illustrates how the R.STACK function is used:

```
CALL NewHandle(1024,myMemoryID%,0,0)

IF R.STACK%(0) = 0 THEN
   myHandle@ = R.STACK@(1)
ELSE
   PRINT "Unable to allocate memory handle, error: ";R.STACK%(0)
END IF
```

Using EXFN instead of CALL

The EXFN reserved word is an alternate form of the CALL statement used to call tool set functions from within an expression. By using EXFN, the result of the function can then be used directly in an expression without having to reference the R.STACK function. The syntax for using EXFN is as follows:

```
EXFN [%|@|&|#|$ ] _functionname (parameter list )
```

Note that the underscore symbol MUST precede the tool set name.

The following example shows how the above example can be rewritten using the EXFN reserved word.

```
myHandle@ = EXFN@_NewHandle(1024,myMemoryID%,0,0)

IF R.STACK%(0) <> 0
  THEN PRINT "Unable to allocate memory handle, error: ";R.STACK%(0)
```

In the example above, the type character @ was used after the EXFN reserved word. This character signifies the function result value is a double integer. The use of a type character after EXFN is strictly optional since TML BASIC *knows* the result type from the function's declaration. In fact, TML BASIC ignores the type character if it is not correct. For example, consider the following variation of the previous example:

```
myHandle@ = EXFN%_NewHandle(1024,myMemoryID%,0,0)
```

Here the % type character is used. However, TML BASIC ignores the type character and *NewHandle* still returns a double integer.

A tool set routine which is defined as a procedure can also be called using the EXFN_ reserved word. In this case, however, the resulting value is the error code returned by the procedure since a procedure does not have a function result.

An Example

The following is a small example program which uses the QuickDraw library to paint the Super Hi-Res screen white and then draw several lines on the screen.

```
LIBRARY "QuickDraw"         'Make the QuickDraw tool set available

GRAF INIT 320               'Initialize the Super Hi-Res screen in 320 mode
GRAF ON                     'Turn the graphics screen on

_ClearScreen(-1)            'Paint the screen white

FOR i% = 0 TO 100 STEP 20
    _MoveTo(30+i%,50+i%)    'Move the QuickDraw pen
    _LineTo(70+i%,90+i%)    'Draw a line
NEXT i%

GET$ Key$                   'Wait for a keypress
GRAF OFF                    'Turn the graphics screen off (text screen on)
END
```

This example shows how simple it is to program simple graphics using the Toolbox. You should now read Chapter 12 to discover more about QuickDraw graphics and then Chapter 13 for information on how to create much more sophisticated programs which use the Toolbox. Example programs in the PART3.EXAMPLES and MORE.EXAMPLES folders also illustrate how to program using the Toolbox.

Chapter 12

Quickdraw Graphics

Drawing to the Screen (and Elsewhere)

Any time your desktop application needs to draw something, it uses the Apple IIGS tool set QuickDraw II (and its extension, QuickDraw II Auxiliary). QuickDraw II is an adaptation and extension of the Macintosh toolbox component *QuickDraw*—it performs similar operations but has been enhanced to support Apple IIGS color.

QuickDraw II allows you to perform graphic operations easily and quickly. QuickDraw draws text in different fonts with styling variations such as italics and boldface. It draws lines and shapes of various sizes and patterns. It can also draw items in a variety of colors or gray scales.

QuickDraw II can draw to the screen or to other parts of Apple IIGS memory. In fact, printing a document with the Print Manager involves using QuickDraw to "draw" your document into a memory buffer used by the Print Manager.

NOTE

For brevity, we'll use the terms *QuickDraw* and *QuickDraw II* synonymously here. Unless otherwise explicitly stated, *QuickDraw* means the Apple IIGS tool sets QuickDraw II and QuickDraw II Auxiliary, not the Macintosh version.

To get our bearings, we'll first consider *where* QuickDraw II draws. Then we'll briefy discuss *how* it draws, and finally look at *what* it draws. The chapter ends with two examples that tie together several of the key ideas.

Where QuickDraw II Draws

The question of *where* QuickDraw II draws involves consideration of Apple IIGS memory (including screen memory) as well as QuickDraw's own internal representation of its drawing universe. These are the main concepts:

- Drawings are stored in Apple IIGS memory as *pixel images,* ordered collections of bytes that represent rectangular arrays of pixels. Screen memory contains a special pixel image—its contents are displayed on the computer's monitor.

- QuickDraw II draws its text and graphic objects on an abstract two-dimensional mathematical surface called the *coordinate plane.* Points on a plane are much easier to visualize and manipulate than addresses in memory. Locations on the QuickDraw II coordinate plane are related to pixel-image memory locations by specific *location information* supplied to QuickDraw.

- Quickdraw draws most objects within the context of *graphic ports.* A port is a complete drawing environment and defines, among other things, a specific part of memory and a specific rectangular area on the coordinate plane where drawing can occur. There can be many open ports at a time—some for drawing to the screen, some for drawing to other parts of memory. Different ports' drawing spaces may be separate from each other or they may overlap.

- QuickDraw II can be made to *clip,* or constrain its drawing, to within limits of arbitrary size, shape, and location.

- By manipulating two independent sets of coordinates (*global coordinates* and *local coordinates*), an application can easily control both what gets drawn inside a port's drawing space and where, on the screen or other pixel image, that drawing space appears.

The Coordinate Plane

QuickDraw locates every action it takes in terms of coordinates on a two-dimensional grid (Figure 12-1). The grid is QuickDraw's *coordinate plane;* coordinates on the plane are integers ranging from –16K to +16K in both the X- and Y-directions. The point (0,0), therefore, is in the middle of the grid. Note also that grid values increase to the right and *downward* on the plane; this is different from what you might be used to, but it is the same direction and order in which video scan lines are drawn.

Distances on the grid are measured in *pixels.* Thus a 10 x 10 "square" on the coordinate plane is equivalent to a rectangle 10 pixels by 10 pixels on the display screen (which would not be a square, of course, because Apple IIGS pixels are not square). Only a very small portion of the coordinate plane can be displayed on the screen at any one time—the plane is 32,000 pixels on a side, whereas the screen can show a maximum of 640 pixels by 200 pixels at a time. Figure 12-1 shows the approximate size of the screen (and user) compared to the coordinate plane.

IMPORTANT

QuickDraw must not be asked to draw outside the coordinate plane. Commands to draw outside this space will produce unpredictable results. They won't generate errors.

Macintosh programmers: This conceptual drawing space is not the same size as that used by QuickDraw on the Macintosh. On the Macintosh, the drawing space is 64K by 64K pixels centered around 0,0, thus making the boundary coordinates -32K,-32k and 32K,32K.

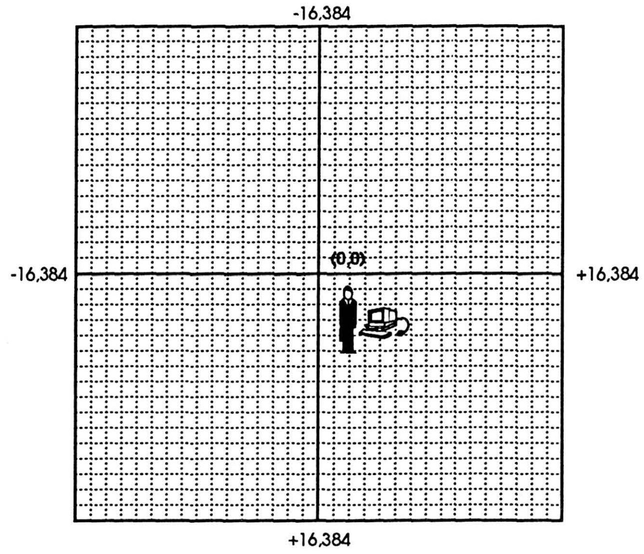


Figure 12-1
The QuickDraw II coordinate plane

To understand how QuickDraw does its drawing, we need to consider how it represents some basic graphic elements. On the coordinate plane, grid lines are considered to be infinitely thin. A point is defined as the intersection of two grid lines, so it also has no dimensions. Pixels, on the other hand, have a definite size; they are thought of as falling *between* the lines of the grid. The smallest element that QuickDraw can draw is a pixel, so if it were to draw a point at the location (3,3) on the coordinate plane, it must draw a single pixel. But which one? Four pixels touch the point. QuickDraw defines the pixel corresponding to each point on the plane as the pixel immediately *below and to the right* of the point. See Figure 12-2.

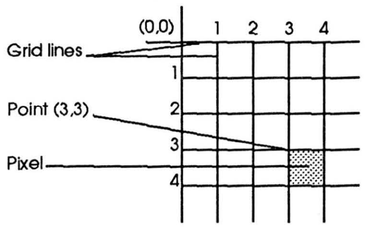


Figure 12-2
Grid lines, points, and pixels on the coordinate plane

Pixel Images and the Coordinate Plane

A *pixel image* is an area of memory that contains a graphic image. The image is organized as a rectangular grid of pixels occupying contiguous memory locations. Each pixel has a value that determines what color in the graphic image is associated with that pixel.

Macintosh programmers: QuickDraw II's pixel images are similar to Macintosh QuickDraw's *bit images*. The major difference is that a pixel is described by more than a single bit.

As described above, QuickDraw II draws to the coordinate plane. However, the coordinate plane is really just an abstract concept. Inside the Apple IIGS drawing actually occurs by modifying pixel images—that is, by modifying the contents of certain memory locations. In particular, drawing something visible on the screen involves modifying the contents of screen memory.

The data structure that ties the coordinate plane to memory is the *LocInfo* (for *location information*) record. The LocInfo record tells QuickDraw where in memory to draw, how the pixel image in that part of memory is arranged, and what its position on the coordinate plane is. In TML BASIC, the LocInfo data structure looks like this:

```
DIM aLocInfoRec!(15)
```

Element(s)	**Value**	**Description**
0	Integer	Scanline control byte (portSCB)
2..5	Double Integer	Pointer to a pixel image (ptrToPixImage)
6..7	Integer	Width
8..15	*Rect*	BoundsRect

The record consists of four fields:

- **portSCB** (a replica of the *scan-line control byte*) tells QuickDraw how many bits per pixel there are in this image—two for 640 mode, four for 320 mode.

  The scan-line control byte and the differences between 640 mode and 320 mode are discussed further under "Drawing in color", later in this section.

- **ptrToPixImage** (or *image pointer*) is the memory address of the image. It points to the first byte of the pixel image, which contains the first (upper-leftmost) pixel.

- **width** (or *image width*) specifies the width (in *bytes*, not pixels) of each line in the pixel image. QuickDraw needs to know this so it can tell where each new row in the image starts. (The image width must be an even multiple of 8 bytes.)

- **boundsRect** (for *boundary rectangle*) is a rectangle that maps the pixel image onto the coordinate plane. The upper-left point in the rectangle corresponds to the first pixel in the image. The lower-right corner of the rectangle describes the extent of the pixel image (as far as QuickDraw is concerned). See Figure 12-3.

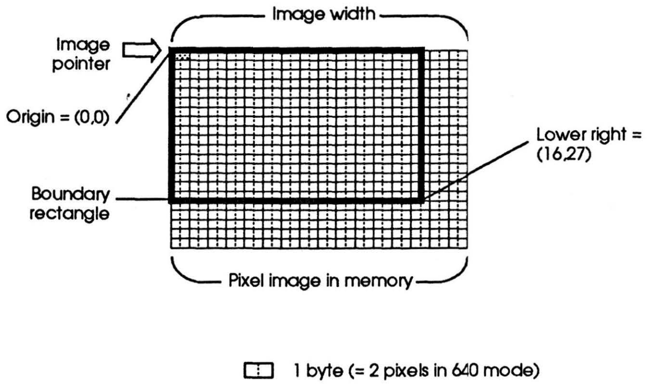


Figure 12-3
Pixel image and boundary rectangle

NOTE

Remember, what separates one pixel image from another is where *in memory* it is stored, not where on the QuickDraw coordinate plane its boundary rectangle happens to be. You can think of each pixel image as having its own private copy of the entire coordinate plane to play with, so that even if two pixel images have overlapping coordinate plane locations, there won't be any conflict between them if they occupy completely different parts of computer memory.

GrafPort, Port Rectangle, and Clipping

Most drawing takes place in conjunction with a data structure called a *GrafPort* (for *graphic port*). Each GrafPort contains a complete specification of a drawing environment, including the location information (LocInfo structure) described above. In addition to the location information, a GrafPort contains three other fields that restrict where drawing in a pixel image can take place: the *port rectangle*, *clipping region*, and *visible region*.

The *port rectangle* (or portRect) is a rectangle on the coordinate plane. Any drawing in a GrafPort occurs only inside its portRect. When you look at a *window* on the screen in a desktop application, its interior (everything but its frame) corresponds to a port rectangle.

Windows are described further in Chapter 13.

The port rectangle can coincide with the boundary rectangle or it can be different. You can think of it as a movable opening, allowing access to all or part of the pixel image. As Figure 12-4 shows, QuickDraw can draw only where the boundary rectangle and port rectangle overlap.

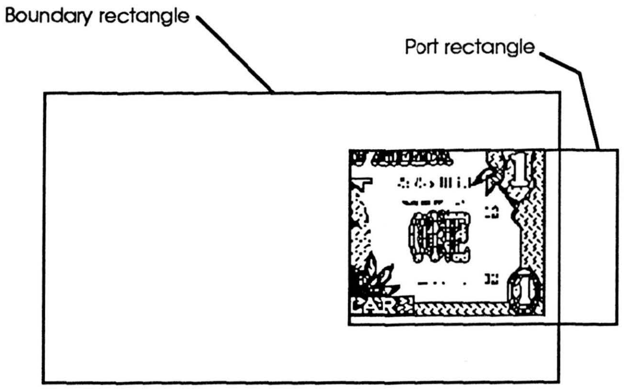


Figure 12-4
Boundary rectangle/port rectangle intersection

The *clipping region* (or clipRgn) is provided for an application to use. When a GrafPort is opened or initialized, the clipping region is set to the entire coordinate plane (effectively preventing any clipping from occuring). The program can use the clipRgn in any way it wants. Any drawing to a pixel image through a GrafPort occurs only inside the clipping region.

The *visible region* (or visRgn) is normally maintained by the Window Manager. An application can have multiple windows on the screen, each one associated with a GrafPort. Windows can overlap, and each port's visible region represents the parts of the window that are visible.

In summary, drawing occurs in a pixel image only in the *intersection* of the boundary rectangle, port rectangle, clipping region, and visible region.

Global and Local Coordinate Systems

Everything is positioned in QuickDraw's universe in terms of coordinates on the plane. However, if you think of multiple open windows on the screen, you can see that there are at least two different ways in which you might want to locate objects:

- You may want to specify where windows appear on the screen (for example, when they are moved).

- You may want to specify where objects appear within windows (for example, when scrolling), independently of where on the screen the windows may be.

The toolbox needs *global coordinates* whenever more than one GrafPort share the same pixel map; the global coordinates tell QuickDraw exactly where every port rectangle is compared to every other one. The global coordinate system for each GrafPort is that in which the boundary rectangle for its pixel map has its *origin* at (0,0) on the coordinate plane. For drawing to the screen, you can think of global coordinates as *screen coordinates*, where the upper-left corner of the screen is the point (0,0).

The *origin* of a rectangle, in QuickDraw II, is its upper-left corner.

However, each port also has its own *local coordinate* system. For example, when drawing into a port it might be more convenient to think in terms of distance from the port rectangle's origin rather than the boundary rectangle's origin. By defining the port rectangle as starting at (0,0), you can base all your drawing commands on distance in from the left edge and down from the top of the portRect.

That's convenient for drawing in a window, but local coordinates are more of a convenience than that. They aren't constrained to a value of (0,0) for the port rectangle origin—you can set them to any coordinate-plane value. Why would you want to? Because of the way drawing commands work.

Suppose you are using a window to display portions of a document that is larger than the port rectangle in size—a fairly common occurrence. You are using drawing commands that draw the entire document, and you know that's no problem because the drawing will be automatically clipped to the port rectangle. But how do you control *which part* of the document shows in your window? You do it by adjusting local coordinates.

All QuickDraw's drawing commands are based on the current port's *local* coordinate system. So if location (0,0) in your GrafPort's local coordinates corresponds to the port rectangle's upper-left corner, any time you draw your document into that port, its upper-left corner will be displayed. If you define your local coordinates differently, different parts of your document will appear in the window. Thus you can think of local coordinates as *document coordinates*—the upper-left corner of the document being viewed in the port has the value (0,0) in local coordinates. See Figure 12-5.

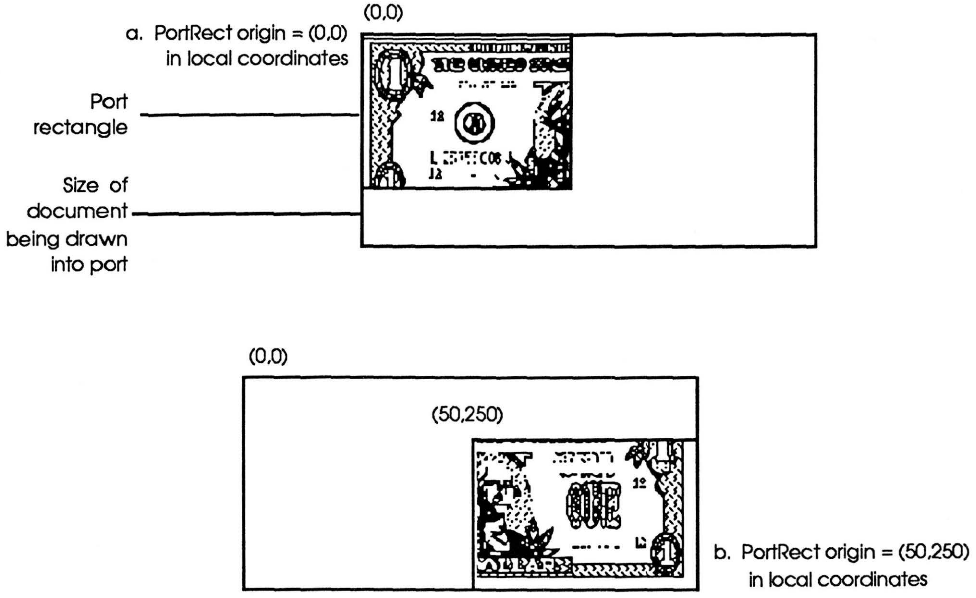

(0,0)
a. PortRect origin = (0,0) in local coordinates
Port rectangle
Size of document being drawn into port
(0,0)
(50,250)
b. PortRect origin = (50,250) in local coordinates

Figure 12-5
Drawing different parts of a document by changing local coordinates

NOTE

When the local coordinates of a GrafPort are changed, the coordinates of the GrafPort's *boundary rectangle* and *visible region* are similarly recalculated, so (as noted) the port will not change its relative position on the screen or in relation to other open ports on the screen.

However, when the local coordinates are changed the GrafPort's *clipping region* and *pen location* are not changed—that is, they appear to shift right along with the image that is being viewed *in* the port. It makes sense to have the pen, which is used to modify the image being viewed, and the clipping region, which is used to mask off parts of the image being viewed, "stick" to it.

Pen location and other pen characteristcs are described next, under "How QuickDraw II Draws".

How QuickDraw II Draws

How QuickDraw II draws any of its objects depends on the drawing environment specified in the current GrafPort. Each GrafPort record includes location and clipping information (described above), information about the graphics pen (described next), information about any text that will be drawn (described under "...And Text Too", later in this section), and other information such as the patterns to draw with.

The Drawing Pen

Each open port has its own drawing *pen*. By means of several characteristics modifiable by the application, the pen controls where and how drawing (of both text and graphics) occurs.

Pen location: The pen has a coordinate-plane location (in local coordinates). The pen location is used for drawing lines and text only—other shapes are drawn independently of pen location.

Pen size: The pen is a rectangle that can have almost any width or height. Its default size is 1 x 1 (pixels). If either the width or height is set to 0, the pen will not draw.

Pen pattern: The pen pattern is a repeating array (8 pixels by 8 pixels) that is used like ink in the pen. Wherever the pen draws, the pen pattern is drawn in the image. The pattern is always aligned with the coordinate plane so that adjacent areas of the same pattern drawn at different times will blend in a continuous manner.

Background pattern: The background pattern is an array similar to the pen pattern. Erasing is the process of drawing with the background pattern.

Drawing mask: The drawing mask is an 8-bit by 8-bit pattern that is used to mask, or screen off, parts of the pattern as it is drawn. Only those pixels in the pattern aligned with an *on* (=1) bit in the mask are drawn. Figure 12-6 shows how a mask affects drawing with a pattern.

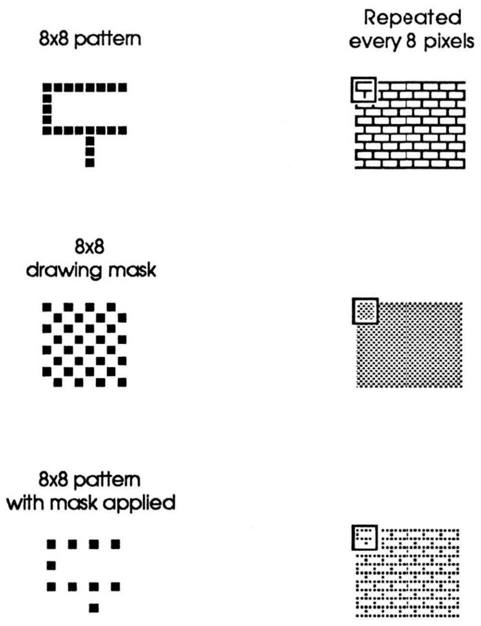


Figure 12-6
Drawing with pattern and mask

Note that drawing with a mask in which every bit has the value 1 is like drawing with no mask at all—all pen pixels are passed through to the image. Likewise, drawing with a mask that is all zeros is like not drawing at all—all pen pixels are blocked.

Pen mode: The pen mode specifies one of eight logical operations (COPY, notCOPY, OR, notOR, XOR, notXOR, BIC and notBIC) that determine how the pen pattern is to affect an existing image. When the pen draws, QuickDraw II compares pixels in the existing image with their corresponding pixels in the pattern, and then uses the pen mode to determine the value of the resulting pixels. For example, with a pen mode of COPY, the existing pixels' values are ignored—a solid black line is black regardless of the image already on the plane. With a pen mode of notXOR, the bits in each pen pixel are inverted and then combined in an exclusive-OR operation with the bits in each corresponding existing pixel. Figure 12-7 shows a rectangle drawn over an existing circle, in both COPY and notXOR mode.

The *Apple IIGS Toolbox Reference* contains further information about pen modes in its section titled "QuickDraw II".

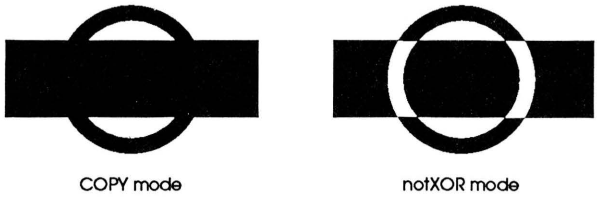


Figure 12-7
How pen mode affects drawing

Basic Drawing Functions

QuickDraw draws *lines* with the current pen size, pen pattern, drawing mask, and pen mode. QuickDraw draws other shapes (*rectangles, rounded-corner rectangles, ovals, arcs, polygons* and *regions*) in five different ways:

QuickDraw's shapes are described next, under "What QuickDraw II Draws".

- **Frame**: QuickDraw draws an outline of the shape, using the current pen size, pen pattern, drawing mask, and pen mode.
- **Paint**: QuickDraw fills the shape, using the current pen pattern, drawing mask, and pen mode.
- **Erase**: QuickDraw fills the shape, using the current background pattern and drawing mask.
- **Invert**: QuickDraw inverts the pixels in the shape, using the drawing mask.
- **Fill**: QuickDraw fills the shape, with a specified pattern and using the drawing mask.

QuickDraw draws *text* as described under "...And Text Too", later in this section.

What QuickDraw II Draws

QuickDraw II can draw a number of graphic objects into a pixel image. It draws text characters in a variety of monospaced and proportional fonts, with styling variations that include italics, boldfacing , underlining, outlining, and shadowing. It draws straight lines of any length, width, and pattern. It draws hollow or pattern-filled rectangles, circles, and polygons. It draws elliptical arcs and filled wedges, irregular shapes and collections of shapes. It also draws *pictures*—combinations of these simple shapes. Figure 12-8 summarizes them.

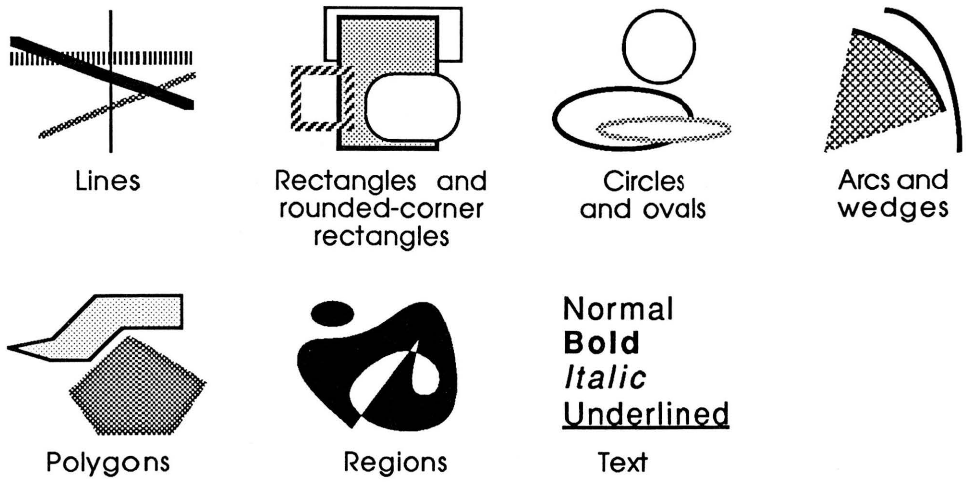


Figure 12-8
What QuickDraw II draws

Points and Lines

A *point* is represented mathematically by its Y- and X-coordinates—two integers. A *line* is represented by its ends—two points, or four integers. Like a point, a line is infinitely thin. When drawing a line, QuickDraw II moves the upper-left corner of the pen along the straight-line trajectory from the current pen location to the destination location. The pen hangs below and to the right of the trajectory, as illustrated in Figure 12-9.

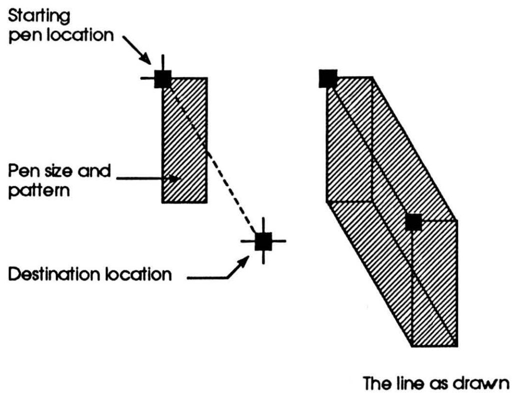


Figure 12-9
Drawing lines

Before drawing a line, you can use QuickDraw calls to set the current pen location and other characteristics such as pen size, mode, and pattern.

IMPORTANT

QuickDraw's data structure that defines a point has the vertical coordinate first: (y,x) rather than (x,y).

Rectangles

A *rectangle* (Figure 12-10) is also represented by two points: its upper-left and lower-right corners. The borders of a rectangle are infinitely thin. Rectangles are fundamental to QuickDraw; there are many functions for moving, sizing, and otherwise manipulating rectangles.

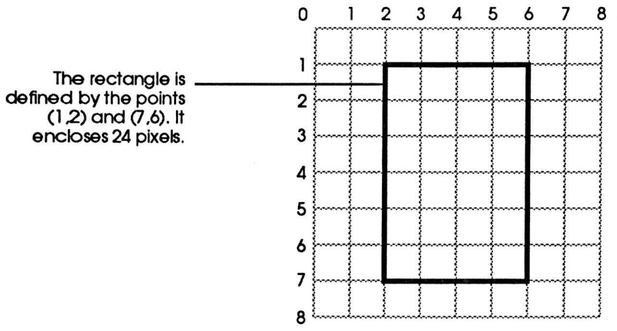

0 1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
The rectangle is defined by the points (1,2) and (7,6). It encloses 24 pixels.

Figure 12-10
A rectangle

The pixels associated with a rectangle are only those *within* the rectangle's bounding lines. Thus the pixels immediately below and to the right of the bottom and right-hand lines of the rectangle are not part of it.

Rectangles may have square or rounded corners. The corners of rounded-corner rectangles are sections of *ovals* (described next); they are specified by an *oval height* and *oval width.*

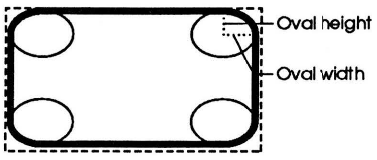


Figure 12-11
Rounded-corner rectangle

IMPORTANT

The QuickDraw data structure that defines a rectangle has coordinates in the following order: top, left, bottom, right. Thus the defining coordinates for the rectangle in Figure 12-10 are (1,2,7,6). This may seem strange, but it is consistent with the (y,x) ordering of points.

Circle, Ovals, Arcs, and Wedges

Ellipses and portions of ellipses form another class of shapes drawn by QuickDraw II. An *oval* is an ellipse, and it is defined just like a rectangle—the only difference is that QuickDraw is told to draw the ellipse inscribed within the rectangle rather than the rectangle itself. If the enclosing rectangle is a square, the resulting oval is a circle.

Pixel shape: Remember, Apple IIGS pixels are not square. A true circle on the screen, or a true square, will have unequal horizontal and vertical dimensions in terms of pixels.

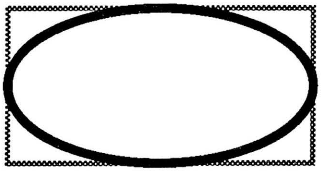


Figure 12-12
Oval

An *arc* is a portion of an oval, defined by the oval's enclosing rectangle and by two angles (the starting angle and the arc angle), measured clockwise from vertical.

If an arc is painted, filled, inverted, or erased, it becomes a *wedge;* its fill pattern extends to the center of the enclosing rectangle, within the area defined by the lines bounding the arc angle.

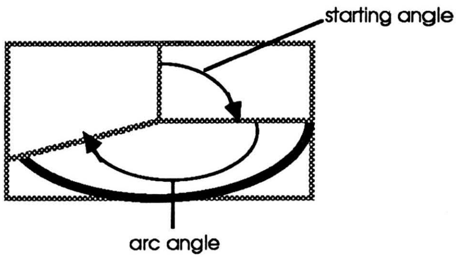


Figure 12-13
Arc

Polygons

A *polygon* is any sequence of connected lines. You define a polygon by moving to the starting point of the polygon and drawing lines from there to the next point, from that point to the next, and so on.

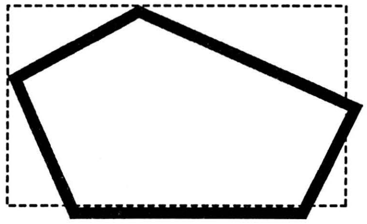


Figure 12-14
Polygon

Polygons are not treated in exactly the same manner as other closed shapes such as rectangles. For example, when QuickDraw II draws (*frames*) a polygon, it draws outside the actual boundary of the polygon, because the line-drawing routines draw below and to the right of the pen locations. When it paints, fills, inverts, or erases a polygon, however, the fill pattern stays within the boundary of the polygon. If the polygon's ending point isn't the same as its starting point, QuickDraw adds a line between them to complete the shape.

Regions

A region is another fundamental element of QuickDraw, one that can be considerably more complex than a line or a rectangle. A *region* can be thought of as a collection of shapes or lines (or other regions), whose outline is one or more closed loops. Your application can draw, erase, move, or manipulate regions just like any other QuickDraw structures.

You can define regions by drawing lines, framing shapes, manipulating existing regions, and equating regions to rectangles or other regions.

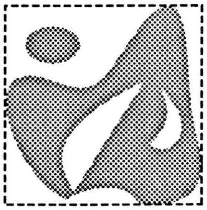


Figure 12-15
Region

Regions are particularly important to the Window Manager, which must keep track of often irregularly shaped, noncontiguous portions of windows in order to know when to activate the windows or what parts of them to update.

Pictures

A *picture* is a collection of any QuickDraw drawing commands. Its data structure consists of little more than the stored commands. QuickDraw plays the commands back when the picture is reconstructed with a DrawPicture call. A complex mechanical drawing produced from an Apple IIGS drafting program might be saved as a single QuickDraw II picture.

...And text too

QuickDraw II doesn't draw graphic images only—it also does all text drawing for desktop applications. As an application programmer, you can easily control the placement, size, style, font, and color of display text with QuickDraw calls.

Your program can provide QuickDraw II with text in a number of formats:

- **Character:** A single ASCII character at a time
- **Pascal string:** A length byte followed by a sequence of ASCII characters
- **C string:** A sequence of ASCII characters terminated by a zero byte
- **Text block:** An arbitrary number of ASCII characters in a buffer

NOTE

TML BASIC automatically converts BASIC strings into Pascal strings for Toolbox routines which have string parameters. See Appendix C for more information.

However it receives the text, QuickDraw II draws it in the same way. It draws each character at the current *pen location,* with the current *font,* using the current *text mode,* with the current character *style,* and using the current *foreground* and *background* colors. After drawing each character, QuickDraw updates the pen location for drawing the next one.

Providing QuickDraw with various fonts and character styles is the job of the Font Manager. The Font Manager is a tool set that supports QuickDraw's character-drawing ability by providing an application with different fonts and styled variations of fonts. If you want to allow the user to choose from all of the fonts available when the application is run, or if you're developing an application that requires a specific font, the Font Manager can help you.

Characters

To help understand just where text appears and how much space it takes up, let's define a few terms. Refer to Figure 12-16.

Text fonts are made up of individual *characters.* A character is represented in memory as a rectangular array of bits, called a *character image,* representing rows and columns of pixels. The *on* (=1) bits are the *foreground pixels;* the *off* bits (=0) are the *background pixels.*

Every character in a font has a baseline. The *base line* is a horizontal line, in the same position for every character in the font. Any foreground pixels of a character image that lie below the base line constitute the character's *descender* (characters like *p* and *q* have descenders). The *ascent line* is the horizontal line just above the top row of a character (including any blanks); the distance from the base line to the ascent line is the font's *ascent,* and is equal to the height of the tallest chartacter in the font. The *descent line* is the line just below the bottom row of the character (including any blanks); the distance from the base line to the descent line is the font's *descent,* and is equal in size to the largest descender in the font.

Each character's *origin* is a point on the baseline that is used to position the character for drawing. This point need not touch any foreground pixels of the character image. When the character is drawn, it is placed in the destination location so that its character origin *coincides with the current pen location.* For many letters, the character origin is located on the left edge of the character image; then, when the character is drawn, its leftmost foreground pixels fall just to the right of the pen location.

The *font height* is the sum of the ascent and descent heights, and it is the same for all characters in a font. The *character width* is the number of pixels the pen position is to be advanced after the character is drawn. It includes the width of the character itself and any needed space between it and the next character to be drawn.

Font height, ascent, descent, character width, and *leading* (the vertical space between lines of text) are needed for calculating string lengths and line spacings when you display text on the screen.

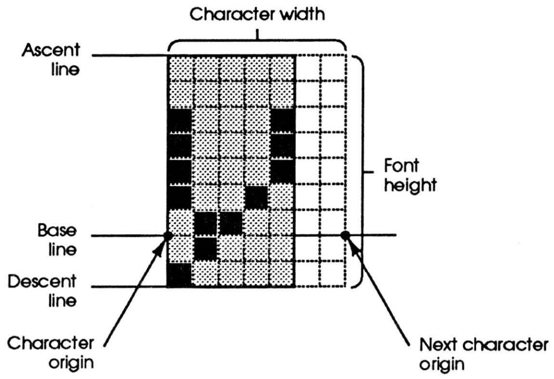


Figure 12-16
A character image

The basic commands necessary to draw characters on the screen are quite simple. The following four commands illustrate how the message "One moment please..." is drawn with white letters and a black background.

```
_SetBackColor(0)                          'Background color = black
_SetForeColor(15)                         'Foreground color = white
_MoveTo(20,20)                            'Move pen to upper left of screen
_DrawString("One Moment Please...")       'Write the message
```

Once the foreground and background colors are set, all that's needed to display a character string is to move the pen to the desired location, and call the QuickDraw routine DrawString.

Fonts

Each collection of related characters is called a *font*. With the font manipulation capabilities of the Font Manager, your Apple IIGS applications can show sophisticated text display in a variety of fonts, sizes, and styles.

The font strike: All the character images making up a font are stored in memory as a *font strike*. A font strike is a long, rectangular array of bits consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code. The character images in the font strike abut each other; no blank columns are left between them.

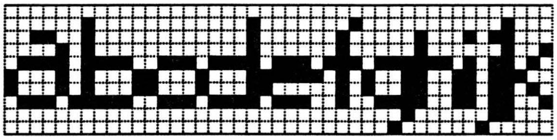


Figure 12-17
Part of a font strike

A given font strike need not contain a character image for every possible ASCII code. The font may leave some characters undefined; these are called *missing characters*. Immediately following the last defined character in the font strike is a character known as the *missing symbol*, which is to be used in place of any missing character. In many fonts the missing symbol is a hollow rectangle; in the Apple IIGS system font, it's a white-on-black question mark. Whenever the QuickDraw II text-handling routines encounter a missing character, they substitute the missing symbol for the character.

Choosing a font: Fonts for the Apple IIGS are grouped into *font families*. Individual fonts within families can have various characteristics, as noted in the following list. When your application requests a font, the Font Manager searches all available fonts and chooses the one which most closely matches the request, in these categories:

- **Name:** Every font family has a name. The name refers to both *plain-styled* characters of all sizes, and any *styled variations,* such as bold or italics.
- **Number:** Every font family has a number, also independent of point size or style modifications. Every family number is unique, and corresponds to a single family name. $0000 represents the system font. Whenever an application requests a font whose family number is not available, the Font Manager substitutes the system font.

- Size: An individual font has a size, described in points. A *point* is a typesetting measure equal to about 1/72nd of an inch. The Font Manager can provide both real and scaled fonts. A real font is one that actually exists on disk at a particular point size. Conversely, a *scaled font* is one that was enlarged or reduced by calculation from a font of a different size. The Font Manager may scale a font from an existing size if the requested size is not available. Real fonts generally have a better screen appearance than scaled fonts.

- **Style:** An individual font also has a style (or combination of styles). The presently defined styles are

  Plain
  Bold
  Italic
  Underline
  Outline
  Shadow

There are two different ways to obtain styled variations of fonts. First, the Font Manager will provide a styled font if one is available—one whose characters are designed with (for example) bold or italic styling. Second, QuickDraw II can *style* a font—that is, it can produce a bold or italicized version of a plain-styled font. In fact, it can produce any combination of the defined styles.

Fonts that are already styled will not be further styled (in the same manner) by QuickDraw II, regardless of the text styling selected. For example, an italic font is not further italicized if that option is selected on a style menu. However, it could be underlined.

Text cannot be underlined unless the font's characters have a descent value (distance between the base line and descent line) of at least 2 pixels. The Apple IIGS system font (Shaston 8) has a descent value of 1, and therfore cannot be underlined.

IMPORTANT

The Font Manager looks for fonts in the subdirectory called FONTS/ in the SYSTEM/ subdirectory on the system disk. This subdirectory must contain all fonts (except the system font) that are to be available to applications. See Appendix C.

Your application can allow the user to select a font by calling the Font Manager routine ChooseFont.

Drawing in Color

The video display hardware of the Apple IIGS includes advanced color capabilities. Although tool calls make it unneccessary for you to manipulate the hardware directly, knowledge of a few background concepts will help you understand the way QuickDraw II manipulates the colors on the screen.

The Apple IIGS offers two Super Hi-Res graphics modes. Both modes have 200 scan lines, but the scan lines differ in horizontal resolution—one mode has 320 pixels (the color of each specified by 4 bits), and the other has 640 pixels (the color of each specified by 2 bits). In changing from 320 mode to 640 mode, the horizontal resolution is doubled at the expense of dividing the color resolution by four.

Both modes use a *chunky pixel* organization (in which the bits for a given pixel are contained in adjacent bits within one byte), as opposed to *bit planes* (in which adjacent bits in memory affect adjacent pixels on the screen). Therefore the 4 bits of a pixel in 320 mode are in the same memory locations as the 4 bits of a pair of adjacent 2-bit pixels in 640 mode.

Colors on the Apple IIGS are determined from *master color values,* which are mathematical combinations of the primary red, blue, and green hues available on a color monitor. A master color value is a 2-byte number. The low-order nibble of the low-order byte controls the intensity of the color blue. The high-order nibble of the low-order byte controls the intensity of the color green. The low-order nibble of the high-order byte controls the intensity of the color red. The high-order nibble of the high-order byte is not used. Figure 12-18 illustrates the format of a master color value.

	Byte 1								Byte 0							
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value:	(not used)				red				green				blue			

Figure 12-18
Master color value format

A 3-digit hexadecimal number can describe each master color, with one digit ($0–$F) for each primary color. Thus a master color value of $000 denotes black, $FFF is white, $00F is the brightest possible blue, $080 is a medium-dark green, and so on. Because each primary color has 16 possible values, a total of 4096 colors are possible. At any one time, the Apple IIGS can display only a small subset of all possible colors. An application specifies its colors by constructing one or more *color tables,* short lists of the available colors for any one pixel.

Color Tables and Palettes

Applications cannot specify pixel colors directly, using master color values. Pixels contain only 2 or 4 bits, and it takes 12 bits to specify a master color value. That's why color tables are necessary. A *color table* is a table of 16 2-byte entries. Each entry in the table is a master color value; any of the 4096 possible color values may appear in any position in the color table.

An application determines the color of a given pixel by specifying an *offset* into the color table. The number of bits used to describe a pixel limits how far into the table it can reach. The colors available to the application, as specified in its color tables, constitute its *palette*. See Figure 12-19.

Pixels in 320 mode are represented in memory by 4-bit integers. For each pixel, that 4-bit value is used as an a offset into a color table. With 4 bits, there are 16 possible pixel values, so the palette in 320 mode is 16 colors—the entire color table.

Pixels in 640 mode are represented in memory by 2-bit integers. With 2 bits, there are 4 possible pixel values to offset into the color table, so the palette in 640 mode consists of only 4 colors. That would seem to leave three-quarters of the color table unused in 640 mode, and severely restrict the use of color, but it's not really so.

In the first place, each 4 adjacent pixels in 640 mode use 4 different parts of the *same* color table; a color table, then, consists of four *mini-palettes*, which needn't have the same sets of master colors. Therefore, although each individual pixel in 640 mode can have one of only four colors, groups of four pixels can have a total of 16 colors from which to choose. How to use this ability to create a large variety of colors is described under "Dithered Colors in 640 Mode", later in this section.

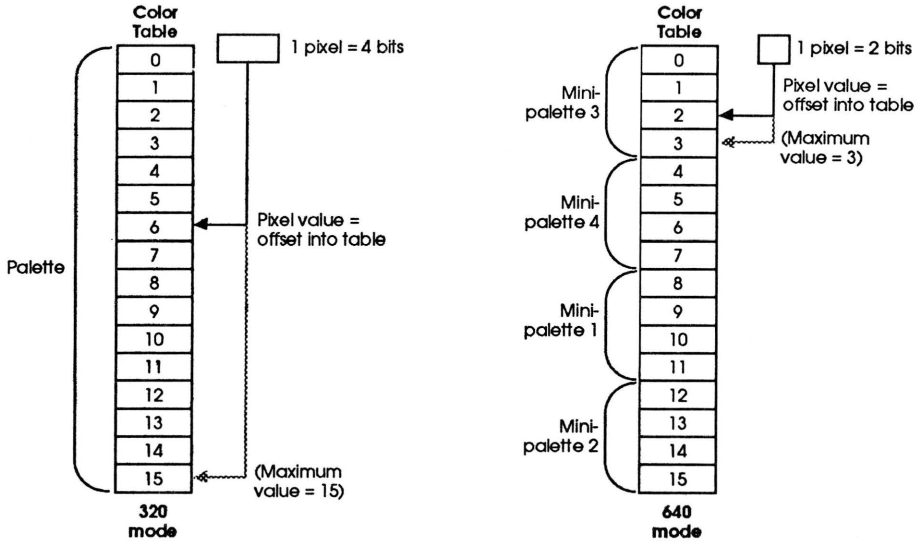


Figure 12-19
Accessing the color table in 320 mode and 640 mode

An application may construct as many as 16 different color tables to choose from. Each of the 200 scan lines in Super Hi-Res graphics can use any one of the 16 tables. For each scan line, a *scan line control byte (SCB)* decides which color table is active. The SCB also controls screen display mode (320 or 640), interrupt mode (whether or not to generate an interrupt during *horizontal blanking*), and *fill mode* (whether or not pixel values of zero can be used to fill areas of color in 320 mode).

Standard Color Palette (320 Mode)

The standard palette (the default color table) for 320 mode is shown in Table 12-1. In the table, *offset* means positon in the color table, and *value* means master color value, the hexadecimal value controlling the fundamental red-green-blue intensities.

Table 12-1
Standard palette—320 mode

Offset	Color	Value
0	Black	000
1	Dark Gray	777
2	Brown	841
3	Purple	72C
4	Blue	00F
5	Dark Green	080
6	Orange	F70
7	Red	D00
8	Beige	FA9
9	Yellow	FF0
10	Green	0E0
11	Light Blue	4DF
12	Lilac	DAF
13	Periwinkle Blue	78F
14	Light Gray	CCC
15	White	FFF

The standard palette was selected because of its flexibility and appearance; we recommend that you use it unless you have a specific need to change it.

Dithered Colors in 640 Mode

As explained above, only four colors are available for each pixel in 640 mode. But when small pixels of different colors are next to each other on the screen, their colors blend. For example, a black pixel next to a white pixel appears to the eye as a larger gray pixel. By cleverly choosing the entries in the color table we can make more colors appear on the screen. This process is called *dithering*.

At the same time, in order to preserve the maximum resolution for displaying text, both black and white must be available for each pixel. This leaves only two remaining colors per pixel to choose from, which seems like a severe restriction. But with dithering, you can have 640-mode resolution for text and still display 16 or more colors, if you are willing to resort to a few simple tricks.

Consider the following byte with four pixels in it:

Bit value	0 1	0 1	0 1	0 1
Pixel number	1	2	3	4

Each pixel has the value 1, which is an index into the *second* place in each of the color table's minipalettes (as shown in Figure 12-19). So pixel 1's color is determined by entry 1 in minipalette 1, pixel 2's color is determined by entry 1 in minipalette 2, and so on. If we use the standard 640-mode color table (shown in Table 12-2) then pixels 1 and 3 will appear blue ($00F), and pixels 2 and 4 will appear red ($D00). The eye will average these colors and see violet.

There are 16 different combinations of values that a pair of pixels can assume in 640 mode, meaning that you can obtain 16 colors by this dithering method. To implement it, just make sure that the pattern you use for drawing or filling consists of a repeating array of 4-bit (= 2-pixel) values.

Table 12-2
Standard palette—640 mode

Offset	Color	Value	(minipalette offset)
0	Black	000	0
1	Blue	00F	1
2	Yellow	FF0	2
3	White	FFF	3
4	Black	000	0
5	Red	D00	1
6	Green	0E0	2
7	White	FFF	3
8	Black	000	0
9	Blue	00F	1
10	Yellow	FF0	2
11	White	FFF	3
12	Black	000	0
13	Red	D00	1
14	Green	0E0	2
15	White	FFF	3

Black and white: Note that the entries in the minipalettes for the standard 640-mode color table are set up so that black and white appear in the same positions in each palette. This arrangement provides pure black and white at full 640 resolution, allowing crisper text display.

Chapter 13
Creating a Desktop Application

In this chapter, the issues and techniques for creating desktop applications using the Apple IIGS Toolbox are discussed. In addition, the TML BASIC language statements and functions for implementing desktop programs are discussed in the context of the GSDEMO.BAS example application.

Desktop applications are one of the most advanced types of programs you can create for the Apple IIGS. Not only are good programming skills required, but a solid understanding of the Apple IIGS Toolbox, especially the desktop tool sets, is a must.

This chapter serves only as an introduction to creating desktop applications. While some attention is given to the details of using particular Toolbox routines, especially those used to create menus and windows, no effort is made to describe how all of the Toolbox routines work. The *Apple IIGS Toolbox Reference* is the definitive reference for the Toolbox, and anyone attempting to write more than the simplest of programs will find this reference absolutely necessary.

Appendix C provides a complete list of the Toolbox libraries provided with TML BASIC, and Chapter 11 describes how to use the libraries in a program.

The Desktop Interface

The Apple Desktop Interface is a collection of human interface guidelines developed by Apple Computer which define the look and feel of applications using the Apple IIGS Toolbox. The "look and feel" of an application is the communication between the computer and the user.

The goal of the guidelines is to create a standard behavior for applications that is accessible, nonthreatening and predictably consistent to the user. Applications which faithfully follow the guidelines will be familiar to the user, and thus more likely to be successful.

The complete set of guidelines are documented in the *Human Interface Guidelines* published by Apple Computer.

Human Interface Guidelines

The following is a brief summary of the principles found in the human interface guidelines. If you plan to write desktop applications, the guidelines should be followed so that regardless of the functionality of your program, it will communicate with the user in a consistent and standard fashion.

Metaphors from the real world. Use concrete metaphors and make them plain, so that users have a set of expectations to apply to computer environments. Whenever appropriate, use audio and visual effects that support the metaphor.

Direct Manipulation. Users want to feel that they are in charge of the computer's activities. They expect their actions to have results, and their tools should provide feedback.

See-and-point instead of remember and type. Users select actions from alternatives presented on the screen. Users rely on recognition, not recall; they shouldn't have to remember anything the computer already knows.

Consistency. Effective applications are both consistent within themselves and consistent with one another. An application should develop a standard mechanism for manipulating related objects. If certain operations are common between different applications (cut, copy, paste), the mechanism for using them should be the same.

WYSIWYG - what you see is what you get. There should be no secrets from the user, no abstract commands that only promise future results. There should be no significant difference between what the user sees on the screen and what is eventually printed.

User-initiated actions. The user, not the computer, initiates and controls all actions.

Feedback and dialog. Keep the user informed. Provide immediate feedback.

Forgiveness. Users make mistakes; forgive them.

Perceived stability. Users feel comfortable in a computer environment that remains understandable and familiar rather than one which changes randomly.

Aesthetic integrity. Visually confusing or unattractive displays detract from the effectiveness of human-computer interactions. Users should be able to control the superficial appearance of their computer workplaces—to display their own style and individuality.

Desktop Elements

To implement the principles defined by the human interface guidelines, Apple has defined two classes of standard interface elements for the Apple Desktop Interface:

Screen elements that define the "look" of the interface,

Human-computer interactions that account for the "feel" of the interface.

Screen elements provide the basic visual context for applications. There are three fundamental screen elements: the desktop, windows and menus (see Figure 13-1) The desktop establishes the metaphor for the interface. It is the "surface" on which the other elements of the interface are placed upon. A window is a frame for viewing the objects that an application manipulates. Windows in the Apple IIGS Finder are used to view files and folders on a disk, while windows in TML BASIC are used for viewing text files containing BASIC source code. Finally, pull-down menus provide the central mechanism for users to direct an application to perform an operation. Menus hide the details of using an application, but, at the same time, makes them quickly and easily accessible.

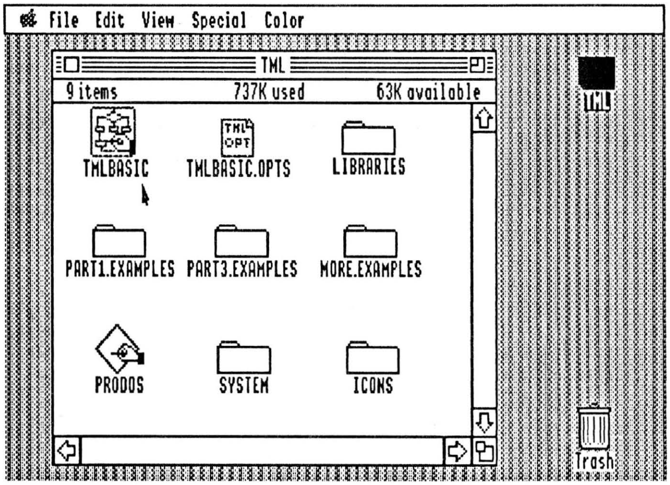


Figure 13-1
The Finder Desktop

Direct manipulation by the user is foremost to human-computer interaction. A pointing device such as the mouse, provides the user direct control of the objects in the desktop. The user can point to objects on the screen, select them with a click of the mouse, move objects, and choose actions which apply to selected objects. The keyboard is also part of the interaction between the user and computer. While the keyboard is generally used for data entry, it usually provides alternatives for moving or selecting objects.

Event-driven Programming

Writing programs which implement the desktop and provide direct manipulation capabilities requires a different style of programming than you may be accustomed.

In the old days, computer programs were executed in batch mode. A stack of cards were fed into the computer, the computer then processed the cards one after the other in the same order every time the program was run. Then came the computer terminal. Users could now interact with a program while it was running. Programs allowed users to send commands that affected the way a program performed. While the user interacted with the program, the program still controlled the choices and the sequence in which operations were performed. The user was still controlled by the program.

Event-driven programming, together with the Apple Desktop Interface, is the complete opposite of these more traditional ways of programming. Event-driven programming means, simply, that the user is in control rather than the program. The basic principle of event-driven programming is that there are many choices available to the user at any one time, and the user controls the sequence of operations performed. For example, the user can invoke operations from pull-down menus, open or close windows, or do work such as word processing or drawing. With few exceptions, any of these operations are available at any one time. That is, a program is *modeless*.

The activities which cause these actions are called *events*. Events can be keypresses, mouse clicks, a disk inserted into a disk drive, data arriving through a serial port, or even events generated by the program itself.

The Main Event Loop

The basic structure of an event-driven program is actually quite simple. The program spends most of its time in a loop called the *main event loop*. The only activity performed in the loop is to wait for an event to occur. When an event occurs, it decides what type of event it is and takes appropriate action.

Figure 13-2 presents a conceptual representation of the execution flow in an event-driven program written in TML BASIC. A program contains a simple loop which repeatedly calls the TASKPOLL statement. Every time the TASKPOLL statement is executed it examines the *event queue* to see if any event has occurred. If so, the event is removed from the queue and handed off to an event-handling subroutine. The subroutine is selected from the *Event Dispatch Table*.

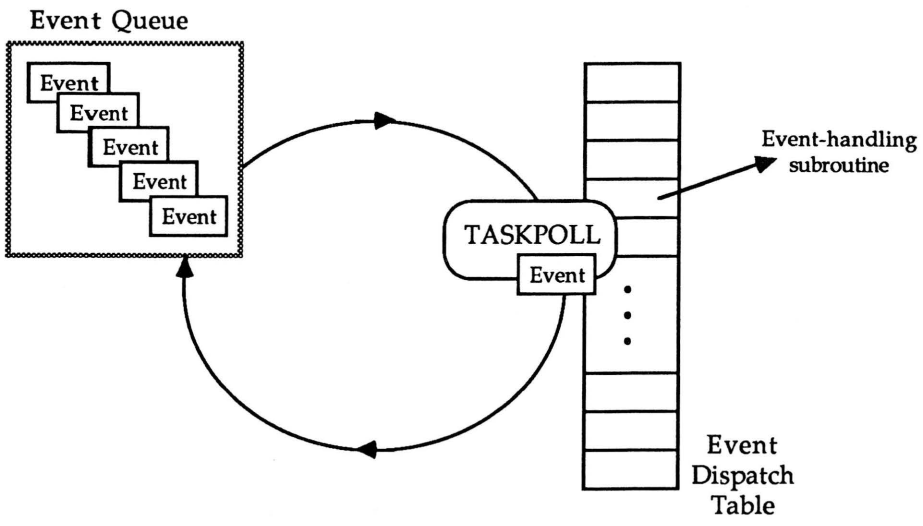


Figure 13-2
Main Event Loop

Event Handling

There are 29 different event types currently defined by the Toolbox. Each of these events is detected by the TASKPOLL statement. Table 13-1 lists each of these events. The first 16 events (numbered 0-15) are the *Event Manager Events*. These are the lowest level events, and are detected by the Event Manager tool set. The events numbered 16 through 28 are the *Window Manager TaskMaster Events*. These are high level events which indicate a mouse-down event in a special part of the desktop, either a window or a menu. They represent the result values from the *FindWindow* toolbox routine.

Table 13-1
Event Manager Event Types

Event Code	Meaning	Description
0	NullEvent	Reported when no other event is available.
1	Mouse-Down	Generated when the user presses the mouse button.
2	Mouse-Up	Generated when the user releases the mouse button.
3	Key-Down	Generated when the user presses any character key on the keyboard or keypad. The character keys include all keys except the Shift, Caps Lock, Control, Option and Apple keys which are modifier keys.

4	*Undefined*	
5	Auto-Key	Generated when the user holds a key down. The auto-key is generated after an initial delay and then at periodic intervals.
6	Update	This is an internally generated event indicating that the contents of a window need to be updated (redrawn).
7	*Undefined*	
8	Activate	This is an internally generated event when a window becomes active or inactive. That is, when a window moves from back to front or from front to back respectively.
9	Switch	Generated when a switch control is pressed.
10	DeskAccessory	Generated when the Classic Desk Accessory menu is invoked via the Control-Apple-Escape key sequence.
11	Device Driver	Generated when a device driver performs a PostEvent due to some circumstance, usually when data transmission has occurred or has been interrupted.
12-15	Application	There are four different application defined events generated. The meaning of these events are defined by the application and entered into the event queue using PostEvent.
16	InDesk	A mouse-down event occurred in the desktop (not in any window).
17	InMenuBar	A mouse-down event occurred in the menu bar and then released over a menu item which was not a desk accessory from the Apple menu or from a menu added by a desk accessory. TaskMaster tracks the mouse until it has been released over a particular menu item, thus selecting it.
18	InSysWindow	A mouse-down event occurred in a system window.
19	InContent	A mouse-down event occurred in the content region of a window.
20	InDrag	A mouse-down event occurred in the drag region of a window.
21	InGrow	A mouse-down event occurred in the grow icon of a window.
22	InGoAway	A mouse-down event occurred in the close box of a window.
23	InZoom	A mouse-down event occurred in the zoom box of a window.
24	InInfo	A mouse-down event occurred in the information bar of a window.
25	*Undefined*	
26	*Undefined*	
27	InFrame	A mouse-down event occurred in the frame of a window.
28	InSpecialMenu	Same as the InMenuBar event, except the selected menu item was one of the Close, Undo, Cut, Copy, Paste or Clear items which did not apply to a desk accessory.

When TASKPOLL detects an event, the event type is used as an index into a special data structure implemented internally by TML BASIC—the *Event Dispatch Table.* The table contains the labels for subroutines which handle events. To enter a label into the table, the EVENTDEF statement is used. For example, the statement:

```
    EVENTDEF 3,HandleKeyDown
```

enters the label *HandleKeyDown* into index 3 of the table. Thus, when TASKPOLL detects a *key-down* event, the *HandleKeyDown* subroutine is automatically called to handle the event. Event-handling subroutines must end with the statement RETURN 0 rather than the normal RETURN statement.

TML BASIC implements a second data structure called the *Menu Item Dispatch Table*. This table allows a program to specify the labels for subroutines which implement each of the menu items in a program. The table contains 128 entries numbered 0 through 127. Subroutine labels are entered into this table using the MENUDEF statement. For example, the statement:

```
    MENUDEF 6,DoAbout
```

enters the label *DoAbout* into index 6 of the table. Thus, when TASKPOLL detects an *InMenuBar* (event 17) or *InSpecialMenu* (event 28) event, and no subroutine label has been specified in the Event Dispatch Table for the event, the appropriate menu item handling subroutine is called from the Menu Item Dispatch Table.

As discussed in the "SetUpMenus" section later in this chapter, every menu item is given a unique integer identifier. The menu item identifiers are in the range 250 to 377 inclusive. Thus, TASKPOLL subtracts 250 from the menu item identifier to obtain the index into the Menu Item Dispatch Table.

To use the TASKPOLL statement, a program must first initialize event processing. This is done with the TASKPOLL INIT statement. The statement has one argument which is the *TaskMask* value. The TaskMask controls which events the TASKPOLL statement is allowed to handle. As noted before, the 29 event types are divided into two classes: Event Manager and TaskMaster. The Event Manager events are the low level events which are always detected. However, the high level TaskMaster events and other operations are controlled by the TaskMask.

The TaskMaster events are detected as a result of further analyzing and tracking a mouse-down event in a menu or window. Since every desktop program contains menus and windows, the event-handling for certain mouse-down events can be handled in a standard way. For example, every-time the mouse button is clicked over the menu bar, the mouse must be tracked in order to pull-down menus and select a menu item. Further, if a desk accessory name is selected, the desk accessory must be opened. The code to perform this operation can be done in a standard way for all programs. TaskMaster (ie TASKPOLL) can perform these operations if requested. The operations TaskMaster should perform are specified by the TaskMask value in the TASKPOLL INIT statement. A mask is the sum of the individual values. Table 13-2 lists the TaskMask values.

Table 13-2
TASKPOLL INIT Mask Values

Value	Description
1	Detect menu keys
2	Perform automatic window updating
4	Perform FindWindow
8	Perform MenuSelect
16	Perform OpenNDA
32	Perform SystemClick
64	Perform DragWindow
128	Perform SelectWindow if mouse down in content
256	Perform TrackGoAway
512	Perform TrackZoom
1024	Perform GrowWindow
2048	Perform automatic scrolling support
4096	Handle special menu items

Thus, a TaskMask value of 8191 (the sum of every value) specifies that TaskMaster should perform all possible high level event processing.

Finally, it is possible to filter certain events from being detected by using the *EventMask* argument of the TASKPOLL statement. Table 13-3 contains the list of event mask values. Again, a complete event mask is obtained by adding the individual values. Of course, a program should generally process all events.

Table 13-3
TASKPOLL Event Mask Values

Value	Description
2	Mouse down
4	Mouse up
8	Key down
32	Auto key
64	Update
256	Activate

512	Switch
1024	Desk Accessory
2048	Device Driver
4096	Application defined #1
8192	Application defined #2
16384	Application defined #3
-32768	Application defined #4

An event mask of -1 indicates that no events should be filtered.

An Example Desktop Application

The remainder of this chapter is dedicated to illustrating the techniques of event-driven programming in the context of the Apple Desktop Interface. The discussions are centered around the example application GSDEMO.BAS. This program is a simple desktop application which has two windows and four menus.

The GSDEMO.BAS program uses the DeskTools library found in the LIBRARIES folder. The DeskTools library contains several procedures and functions which implement operations that most desktop applications require. The complete source code to the library is included so that you can change their behavior as appropriate for your own applications.

Before continuing, you should compile this program and experiment with its features so that you can better appreciate the code necessary to create each element of the program.

The DeskTools Library

Several operations related to desktop applications must be implemented in the source code of every desktop program. These include the loading and initialization of the Apple IIGS Toolbox tool sets used by the application, creating menus and windows and finally, shutting down the tool sets when the program is complete. Since these operations are nearly identical in every program, TML BASIC includes the complete source code to a library of procedures and functions which implement these tasks. This library is the DESKTOOLS.BAS library found in the LIBRARIES folder of the distribution disk.

The following is a list of the procedures and functions declared in the DeskTools library:

```
DEF LIBRARY DeskTools

   DEF PROC StartUpTools(ScreenMode%,LoadPrintTools%)
   DEF PROC ShutDownTools

   DEF PROC StdAppleMenu
   DEF PROC StdEditMenu
   DEF PROC StdFileMenu(FullMenu%)
   DEF PROC DrawMenus

   DEF FN   StdWindow@(Left%,Top%,Right%,Bottom%,Title@,DrawingProc@)
   DEF FN   StdDialog%(Msg1$,Msg2$,NumButtons%)

END LIBRARY
```

The complete source code for this library is found in the file DESKTOOLS.BAS. The procedures and functions in this library are central to the implementation of the GSDEMO.BAS program. As such, they are discussed in the context of the GSDEMO.BAS program throughout the next several sections. Because TML BASIC libraries can be used in any program by using the LIBRARY statement, you will find the routines in this library invaluable when creating your own desktop programs.

In fact, you should create your own libraries which implement common routines among your applications. For example, a library which implements printing would be an extremely useful library.

Writing a Desktop Application

The main part of most desktop programs consists of only six procedure calls. They are the following:

```
PROC StartUpTools(320,0)
PROC SetUpMenus
PROC SetUpWindows
PROC SetUpEventTables
PROC MainEventLoop
PROC ShutDownTools
```

The *StartUpTools* and *ShutDownTools* procedures are responsible for loading, initializing and then shutting down the Toolbox tool sets used in a program. The *SetUpMenus* and *SetUpWindows* procedures are used to create the application's menus and windows. The *SetUpEventTables* procedure enters subroutine labels into the Event Dispatch Table and Menu Item Dispatch Table. And finally, the *MainEventLoop* procedure detects and processes events.

The remaining six sections of this chapter explore the operations of each of these procedures as implemented in the GSDEMO.BAS program and the DESKTOOLS.BAS library. While the examples are specific to these source code files, the discussion is applicable to any desktop program.

The StartUpTools Procedure

StartUpTools is always the first procedure to be executed in an application which uses the Apple IIGS Toolbox. The procedure is responsible for loading and initializing every tool set used by the application. As discussed in Chapter 11, not all of the tool sets reside in the Apple IIGS read-only memory (ROM), but instead on the system disk. These disk-based tool sets must be loaded to random access memory (RAM) before they can be used.

The Apple IIGS Toolbox currently contains 28 tool sets, however, most applications use only a subset of these tools. In fact, most desktop applications use only the following 12 tool sets:

Memory
Miscellaneous Tools
QuickDraw
Event Manager
Window Manager
Control Manager
Menu Manager
Line Edit
Dialog Manager
Standard File
Scrap Manager
Desk Manager

Programs that use the Print Manager for printing documents also use four additional tool sets:

QuickDraw Auxiliary
List Manager
Font Manager
Print Manager

Of course, any particular program may use a subset of these tool sets or other tool sets not listed here, most notably the sound related tool sets.

The following is the source code for the StartUpTools procedure from the DeskTools library which illustrates the technique for loading and initializing the tool sets.

```
DEF PROC StartUpTools(ScreenMode%,LoadPrintTools%)

   'Save the startup parameters in globals
   svScreenMode% = ScreenMode%
   svLoadPrintTools% = LoadPrintTools%

   'Initialize the graphics screen
   GRAF INIT Mode%
   GRAF ON

   'Give a message while waiting for tools to load and start
   _MoveTo(40,40)
   _SetBackColor(0)
   _SetForeColor(15)
   _DrawString("Please wait, starting tools...")

   'Load the standard tools to memory (also load TML BASIC .LIB files)
   LIBRARY LOAD "Memory"
   LIBRARY LOAD "MiscTool"
   LIBRARY LOAD "QuickDraw"
   LIBRARY LOAD "Event"
   LIBRARY LOAD "Window"
   LIBRARY LOAD "Control"
   LIBRARY LOAD "Menu"
   LIBRARY LOAD "LineEdit"
   LIBRARY LOAD "Dialog"
   LIBRARY LOAD "StdFile"
   LIBRARY LOAD "Scrap"
   LIBRARY LOAD "Desk"

   'Load the printing tools if requested
   PrintToolsLoaded% = PrintTools%
   IF PrintTools% THEN
      LIBRARY LOAD "QDAux"
      LIBRARY LOAD "List"
      LIBRARY LOAD "Font"
      LIBRARY LOAD "Print"
   ELSE
      LIBRARY " QDAux"
      LIBRARY "List"
      LIBRARY "Font"
      LIBRARY "Print"
   END IF

   'Start the memory manager
   AppMemoryID% = EXFN_MMStartUp

   'Allocate 10 pages of memory in bank 0 for tool set globals
   ' (1 page = 256K bytes)
   ToolZeroPageH@ = _NewHandle(6*256,AppMemoryID%,-16379,0)
   ToolZeroPageP@ = FN Deref(ToolZeroPageH@)
   ToolZeroPage%  = EXFN_LoWord(ToolZeroPageP@)
```

```
      'Start the standard desktop tools
      _MTStartUp
      _WindStartUp(AppMemoryID%,ToolZeroPage%)
      _CtlStartUp(AppMemoryID%,ToolZeroPage%+256)
      _MenuStartUp(AppMemoryID%,ToolZeroPage%+512)
      _LEStartUp(AppMemoryID%,ToolZeroPage%+768)
      _DialogStartUp(AppMemoryID%,ToolZeroPage%+1024)
      _SFStartUp(AppMemoryID%,ToolZeroPage%+1280)
      _ScrapStartUp
      _DeskStartUp

      'Start the printing tools if requested
      IF PrintTools% THEN
         _QDAuxStartUp
         _ListStartUp
         _FMStartUp(AppMemoryID%,ToolZeroPage%+1536)
         _PrintStartUp(AppMemoryID%,ToolZeroPage%+1792)
      END IF

      'Draw the desktop
      _Refresh(0)

      'Initialize and display the mouse cursor
      _InitCursor
      _ShowCursor
   END PROC StartUpTools
```

The *StartUpTools* procedure is written so that it can be used by most programs that use the Toolbox. As such, it has two parameters. The first parameter indicates whether the desktop should be initialized in 320 or 640 mode, and the second indicates whether the printing tool sets are necessary.

The LIBRARY statement is used to load the required tool sets to memory. The LOAD clause is added after the reserved word LIBRARY to indicate the code necessary to load the tool set to memory should be generated. Note, the printing tool sets are only loaded if the parameter *LoadPrintTools%* is non-zero.

After the tool sets have been loaded, they must be initialized. This is done by calling the *StartUp* procedure for each tool set. The order in which the tool sets are initialized is **very important.** The order shown in the *StartUpTools* procedure is the required order for proper initialization. If additional tool sets are used, they should be started **after** those listed in the *StartUpTools* procedure. Most of the *StartUp* procedures require the application's *MemoryID* and a block of memory to use for storing its global variables. The memory required to start several of the tool sets must be allocated in bank 0 of the Apple IIGS memory as indicated by the parameters to *NewHandle.*

The final operation in this procedure is to draw the desktop and display the mouse cursor.

The ShutDownTools Procedure

The ShutDownTools procedure is always the last procedure called by a desktop application. The procedure is responsible for informing the Toolbox that the application is finished using each of the tool sets it initialized in the StartUpTools procedure, deallocating the memory used by those tool sets, and turning off the graphics screen.

To signal the Toolbox that an application has finished using a tool set, its *ShutDown* procedure is called. Again, the order in which the tool sets are shut down is important, and should be the reverse order in which the tool sets were started.

The following is the ShutDownTools procedure from the DeskTools library.

```
DEF PROC ShutDownTools
'This procedure is used to shut down each of the Apple IIGS
' Toolbox tool sets which were started by the procedure
' StartUpTools

   GRAF OFF

   IF PrintToolsLoaded% THEN
      _PMShutDown
      _FMShutDown
      _ListShutDown
      _QDAuxShutDown
   END IF

   _DeskShutDown
   _ScrapShutDown
   _SFShutDown
   _DialogShutDown
   _LEShutDown
   _MenuShutDown
   _CtlShutDown
   _WindShutDown
   _EMShutDown
   _MTShutDown

   _DisposeHandle(ToolsZeroPageH@)

   _MMShutDown(AppMemoryID%)
END PROC ShutDownTools
```

The first statement in the procedure is GRAF OFF. This statement turns off the super hi-res graphics screen and is equivalent to calling the corresponding QuickDraw procedure.

Note that DisposeHandle is called to deallocate memory before the MMShutDown routine is called. Obviously, it would not make much sense to deallocate memory (a Memory Manager routine) after shutting down the Memory Manager tool set.

The SetUpMenus Procedure

As stated earlier, pull-down menus are one of the fundamental *screen elements* of the Apple Desktop Interface. Pull-down menus consist of three components: the menu bar, the menu titles, and the menu items.

The menu bar is the area across the top of the screen which contains each of the individual menu titles. Each menu title represents a different pull-down menu. Three menu titles are considered standard, and should be present in every desktop application. They are the Apple, File and Edit menus, and should appear in that order as the first menus in the menu bar. Menu titles specific to the application appear to the right of these menus. Finally, menu items are the list of phrases contained in each menu. The menu items correspond to the operations available in an application. If a menu item is dimmed (non-selectable), the operation is not currently available.

The Apple menu normally contains an About... menu item followed by a list of the desk accessories that are currently installed on the system. The menu changes as the user installs new desk accessories or deletes an existing one. The File menu contains operations related to creating, opening, closing and printing documents. At a minimum, the File menu contains the Quit menu item for exiting the application. The Edit menu contains the standard *Clipboard* editing operations. The Edit menu should always contain the Undo, Cut, Copy, Paste and Clear menu items. Even if the application does not support them, they should be included for desk accessories. Of course, application specific editing operations such as Select All or Show Clipboard can be added to the Edit menu.

Every menu and menu item must have an *identifier*. The identifier is a unique integer value which the Menu Manager uses to identify each menu and menu item. Menu identifiers are number from 1 going left to right across the menu bar. Menu item identifiers are numbered in the range 250 to 377. Certain menu item identifiers are reserved. Table 13-4 lists the reserved menu item identifiers.

Table 13-4
Menu Item Identifiers

Value	Meaning	
250	Undo	Menu Manager reserved values
251	Cut	
252	Copy	
253	Paste	
254	Clear	
255	Close	
256	About...	TML BASIC DeskTools values
257	New	
258	Open...	
259	Save	
260	Save As...	
261	Chooser...	
262	Page Setup...	
263	Print...	
264	Quit	
265	*First application specific identifier*	

To create a menu, the Menu Manager *NewMenu* function is used. The menu is defined using a *menu string* parameter. A menu string contains the name of the menu title followed by the names of one or more menu items. Associated with each name are a collection of *attributes* which define the appearance of the items as well as its *item identifier*. The following menu string is used in the *StdAppleMenu* procedure from the DeskTools library to define the Apple menu:

```
    MenuStr$ = ">>@\XN1\0==About...\N256\0==-\N377D\0."
```

As you can see, the menu string is a bit cryptic. The menu title, which appears first in the menu string, is preceded by two greater than symbols (>>), followed by the menu title name, its attributes and finally, a zero byte.

Each menu item is preceded by two equal symbols (==), followed by the menu item name, its attributes and finally, a zero byte. The last character in the menu string must always be a period (.). In TML BASIC, the null character (a zero byte) is created by using the backslash character followed by a zero (\0).

Since it is impossible to type in the name of the colored Apple symbol, the @ symbol is used instead. The letters "\XN1" are the attributes for the menu title. Similarly, the attributes for the "About..." menu item are "\N256". The attributes in menu strings use special codes recognized by the *NewMenu* function. Table 13-5 shows the legal attribute characters for menu strings. Any combination of attributes may be used, however the N or H attribute must always be specified in order to define the menu item identifier.

Table 13-5
Menu Item Attributes

\	Beginning of special attribute characters
*	Followed by a primary, then alternate character to be used as a keyboard equivalent.
B	Bold the menu title
C	Followed by a character to mark the item
D	To dim the item (disable the item)
H	Hexidecimal menu item identifier follows
I	Italicize the menu title
N	Decimal menu identifier follows (between 256 and 3xx)
U	Underline the menu title
V	Places a dividing line under the item without using a separate item
X	Use color replace, and not XOR highlighting.

Because the Menu Manager maintains pointers back into the application where the menu strings are stored, it is required that the menu strings be stored as global variables. Further, the storage for the global variables cannot move during program execution. Since string data is stored in a string pool which can move from time to time, the only alternative is to store menu strings in structure array variables. The SET statement is used to assign a string value into a structure array. For example:

```
DIM AppleMenuStr!(38)
MenuStr$ = ">>@\XN1\0==About...\N256\0==-\N377D\0."
SET(AppleMenuStr!(0)) = ^MenuStr$
```

After a menu is created using the *NewMenu* function, it is added to the menu bar by calling the *InsertMenu* procedure. Menus are insterted into the menu bar in the *reverse* order in which they appear on the screen. After all of the menus have been defined, the menu bar is drawn. Consult the procedures *StdEditMenu, StdFileMenu* and *SetUpMenus* for further examples of creating menus.

The SetUpWindows Procedure

Windows are the third fundamental *screen element* of the Apple Desktop Interface. A window is a frame which presents information. Windows can be of any size or shape, and there can be multiple overlapping windows on the desktop.

Within the frame of a window are several elements. Figure 13-3 illustrates these elements.

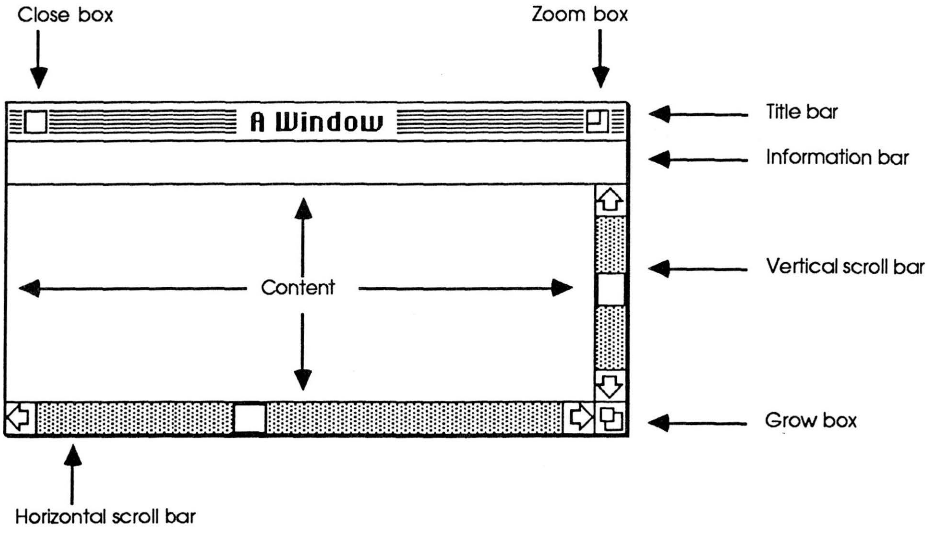


Figure 13-3
Window Elements

Note that not all windows necessarily have each of these elements. Some windows, such as dialogs, have only a frame and the content, others may contain just a title and a scroll bar, while others may of course contain every element. The following is a short description of each window element:

- The title bar displays the window's title, and can hold the close and zoom boxes. It can also be the drag region for moving the window.
- The close box is used to remove the window from the screen
- The zoom box is used to make the window grow to its maximum size and then return it to its previous size.

- The vertical scroll bar allows the user to scroll vertically through the data in the window.

- The horizontal scroll bar allows the user to scroll horizontally through the data in the window.

- The grow box is used to change the size of the window.

- The information bar is used to display information which is not affected by the scroll bars.

In order to create a window on the desktop, the Window Manager function *NewWindow* is used. The function has a single parameter which fully describes the components and behavior of the window. While only a single function is required to create the window, its parameter is very complex. The parameter is a pointer to a NewWindowParamBlk. The definition of a NewWindowParamBlk is as follows (from Appendix C):

```
DIM aNewWindowParamBlk!(73)
```

Element(s)	Value	Description
0..1	Integer	Number of bytes in NewWindowParamBlk (=74)
2..3	Integer	Bit vector that describes the window
4..7	Double Integer	Pointer to window's title : *String*Ptr
8..11	Double Integer	Application RefCon
12..19	*Rect*	Size and position of content when zoomed
20..23	Double Integer	Pointer to window's color table : *WindowColorTbl*Ptr
24..25	Integer	Content's vertical origin
26..27	Integer	Content's horizontal origin
28..29	Integer	Entire height of document
30..31	Integer	Entire width of document
32..33	Integer	Maximum height of content allowed by GrowWindow
34..35	Integer	Maximum width of content allowed by GrowWindow
36..37	Integer	Number of pixels to scroll vertically for arrows
38..39	Integer	Number of pixels to scroll horizontally for arrows
40..41	Integer	Number of pixels to scroll vertically for page
42..43	Integer	Number of pixels to scroll horizontally for page
44..47	Double Integer	Information bar refcon
48..49	Integer	Height of information bar
50..53	Double Integer	Address of standard window definition procedure : *Proc*Ptr
54..57	Double Integer	Address of information bar procedure : *Proc*Ptr
58..61	Double Integer	Address of content update draw procedure : *Proc*Ptr
62..65	*Rect*	Starting position and size of window
66..69	Double Integer	Window's starting plane
70..73	Double Integer	Address of memory to use for window record

One of the most important fields in this data structure is the *window frame bit vector* stored in elements 2 and 3. The bit vector is used to indicate the type of window frame to draw and what elements to create for the window. The definition for each bit in the bit vector follows:

bit 0	1=frame highlighted, 0=unhighlighted
bit 1	1=currently zoomed, 0=not zoomed
bit 2	1=record was allocated, 0=record provided by application
bit 3	1=control's state is independent, 0=inactive window has inactive controls
bit 4	1=window has information bar, 0=no information bar
bit 5	1=currently visible, 0=invisible
bit 6	1=mouse down in content reported even when used to activate window
bit 7	1=title bar is drag region, 0=no drag region
bit 8	1=zoom box in title bar, 0=no zoom box
bit 9	1=GrowWindow and ZoomWindow won't change the origin
bit 10	1=grow box in window, 0=no grow box
bit 11	1=window frame has horizontal scroll bar, 0=no horizontal scroll bar
bit 12	1= window frame has vertical scroll bar, 0=no vertical scroll bar
bit 13	1=alert type window frame, 0=document type window frame
bit 14	1=dose box in title bar, 0=no close box
bit 15	1=title bar, 0=no title bar

To create a window then, all that is necessary is to declare a structure array variable and assign the appropriate values into the structure using the SET statement. An example of this is found in the *StdWindow@* function declared in the DeskTools library.

The following paragraphs examine how the *StdWindow@* function creates the NewWidowParamBlk. In particular, the definition of the window frame bit vector and the drawing procedure. For detailed information about the NewWindowParamBlk, reference the "Window Manager" chapter of the *Apple IIGS Toolbox Reference.*

The following statement is used to assign the window frame bit vector in the StdWindow function:

```
    SET(myWind!(2)) = CONV%(-8800)
```

The CONV% function is used to ensure that the SET statement assigns two bytes into the structure variable as an integer (see the description of the SET in Chapter 10). The value -8800 is equivalent to the binary value "1101 1101 1010 0000". Thus, based on the definition of the *window frame bit vector* given above, the window is defined as follows:

bit 0	0	unhighlighted
bit 1	0	not zoomed
bit 2	0	record provided by application
bit 3	0	inactive window has inactive controls
bit 4	0	no information bar
bit 5	1	currently visible
bit 6	0	mouse down in content not reported when inactive
bit 7	1	title bar is drag region
bit 8	1	zoom box in title bar
bit 9	0	GrowWindow and ZoomWindow change the origin
bit 10	1	grow box in window
bit 11	1	window frame has horizontal scroll bar
bit 12	1	window frame has vertical scroll bar
bit 13	0	document type window frame
bit 14	1	close box in title bar
bit 15	1	title bar

The second statement of significant interest is the one which assigns the address of the window's update drawing subroutine.

```
SET(myWind!(58)) = UpdateProc@
```

This statement defines the address of the subroutine which draws the content of the window. For example, the second window in the GSDEMO.BAS example draws the message "TML BASIC is Great!" several times. The following subroutine is responsible for printing the message.

```
DrawWindow2:
   FOR i% = 1 to 10
      _MoveTo(i%*11+20,i%*9+10)
      _DrawString("TML BASIC is Great!")
   NEXT i%
   RETURN 0
```

To obtain the *address* of this subroutine, its label is first entered into one of the latter 32 entries in the Event Dispatch Table using the EVENTDEF statement. For example:

```
EVENTDEF 63,DrawWindow2
```

Then the address is obtained using the EXEVENT@ function. For example:

```
UpdateProc@ = EXEVENT@(63)
```

This subroutine is then automatically called by TASKPOLL any time the content of the window needs to be redrawn.

Study the *StdWindow@* function in the DeskTools library and the other desktop applications in the MORE.EXAMPLES folder for more information regarding the creation of windows.

The SetUpEventTables Procedure

The *SetUpEventTables* procedure is responsible for entering labels for event-handling subroutines into the Event Dispatch Table and Menu Item Dispatch Table.

As described earlier in the "Event Handling" section, the TASKPOLL statement transfers control to event-handling subroutines automatically when an event is detected. It does this by indexing the Event Dispatch Table and Menu Item Dispatch Tables with the event type to locate the event-handling subroutine.

The GSDEMO.BAS application is a simple desktop application which relies upon TASKPOLL to handle most events in the standard fashion. Recall that this is done by setting the TaskMask to 8191 in the TASKPOLL INIT statement. However, GSDEMO.BAS does implement the *InGoAway* event, and seven menu items.

The *InGoAway* event is implemented by the *HandleGoAway* subroutine. Since the *InGoAway* event is event number 22 (see Table 13-1), the *HandleGoAway* label is entered into the 22nd element of the Event Dispatch Table using the EVENTDEF statement.

```
EVENTDEF 22,HandleGoAway
```

The following seven menu items are implemented in GSDEMO.BAS: About, Quit, Window #1, Window #2, Rects, Ovals, and Round Rects. Menu items are implemented by NOT entering any subroutine label in the Event Dispatch Table for event numbers 17 (InMenuBar) and 28 (InSpecialMenu), but instead entering menu item handling subroutine labels in the Menu Item Dispatch Table. This is done with the MENUDEF statement. The index position used for a menu item is its menu item identifier minus 250. Thus, the following statements are used to enter the menu item handling subroutine labels into the Menu Item Dispatch Table:

```
MENUDEF 6,DoAbout          'Menu identifier #256
MENUDEF 14,DoQuit          'Menu identifier #264
MENUDEF 15,DoWindow1       'Menu identifier #265
MENUDEF 16,DoWindow2       'Menu identifier #266
MENUDEF 17,DoRects         'Menu identifier #267
MENUDEF 18,DoOvals         'Menu identifier #268
MENUDEF 19,DoRRects        'Menu identifier #269
```

Subroutines which implement event-handling must end with the RETURN 0 variation of the RETURN statement. This is because the TASKPOLL statement called the subroutine and not the GOSUB statement. As such, the conventions for calling an event-handling subroutine are different.

The MainEventLoop Procedure

The heart of every event-driven application is the *MainEventLoop* procedure. This is the procedure responsible for detecting events such as a mouse-down, a key-down, a menu-selection, window activates and updates, etc. When an event is detected, it dispatches control to the appropriate subroutine to handle the event. While this sounds like a rather complicated procedure, it is actually quite simple.

The *MainEventLoop* procedure consists of a loop which repeatedly calls the TASKPOLL statement to detect events. When an event is detected, the TASKPOLL statement automatically calls the appropriate event-handling subroutine as specified in the Event Dispatch Table. The following source code is the *MainEventLoop* procedure from the GSDEMO.BAS program:

```
DEF PROC MainEventLoop
   Quit% = 0
   DO
      PROC CheckMenus
      TASKPOLL -1
   UNTIL Quit%
END PROC
```

The first statement assigns the value zero to the global variable *Quit%*. The *Quit%* variable is set to a non-zero value when the user has selected the Quit menu item from the File menu. This is done in the subroutine *DoQuit*. When the value of *Quit%* becomes non-zero, the loop terminates.

Note that the loop does not contain a call to the subroutine *DoQuit*. Instead, it is automatically called by the TASKPOLL statement when an *InMenu* event is detected which selects the Quit menu item. The subroutine is automatically called because the *SetUpEventTables* procedure entered its label into the Menu Item Dispatch Table.

Like the *DoQuit* procedure, the other event-handling subroutines whose labels are in the Event Dispatch Table, or the Menu Item Dispatch Table, are automatically called by the TASKPOLL statement when its corresponding event is detected.

In many cases, the event-handling subroutine needs to know more about the event which has occurred. For example, an application which draws in the content a window must know *where* a mouse-down event occurred so that it can draw at the indicated location. Information about an event can be obtained using the TASKREC% and TASKREC@ functions. These functions return an integer or double integer value from the *TaskRecord* data structure. The TaskRecord is an internal TML BASIC variable which is declared as an Event Manager Event Record. The definition of an Event Record follows (from Appendix C):

```
DIM anEventRecord!(19)
```

Element(s)	**Value**	**Description**	
0..1	Integer	(what)	Event code specifying which event occurred
2..5	Double Integer	(message)	Event message which has additional information about event
6..9	Double Integer	(when)	Number of ticks since startup
10..13	*Point*	(where)	Mouse location where event occurred
14..15	Integer	(modifiers)	Modifier flags
16..17	Double Integer	(Task Data)	Task Data for Task Master
18..19	Double Integer	(Task Mask)	Task Mask for Task Master

Each of the TASKREC functions has an integer parameter. The parameter specifies a word (2 bytes) offset into the TaskRecord. Thus, to determine the location of the mouse for a mouse-down event, *TASKREC@(5)* is called to return the Point field from the TaskRecord as a double integer value. Of course, *TASKREC%(5)* and *TASKREC%(6)* can be called to return the individual horizontal and vertical components of the Point.

The meaning of each field depends upon the event type returned. Table 13-6 lists the meanings of the TaskRecord fields.

Table 13-6
TaskRecord Fields

what	Indicates which of the event types occurred.
message	Contains information specific to the event that has occurred.

Mouse-down	Button number in low order word
Mouse-up	Button number in low order word
Key-down	ASCII code in low-order byte
Auto-key	ASCII code in low-order byte
Activate	Pointer to window to activate (deactivate)
Update	Pointer to window to update
Device-driver	Defined by the device driver
Application	Defined by the application

when	Is the time that the event occurred. The time is given in the number of ticks (1 tick is 1/60th of a second) that have elapsed since you booted the Apple IIGS.
where	Specifies the location of the mouse when the event occurred. The location is given in global coordinates.
modifiers	Offers more specific information when appropriate. Each bit in this field signifies a different piece of information. For example, certain bits indicate whether the Shift, Option, Apple or Control keys were pressed.
TaskData	This field contains the menu identifier and menu item identifier for the InMenuBar and InSpecial events. For all other TaskMaster events, this field contains the Window Pointer of the effected window.
TaskMask	This field is contains the TaskMask value specified in the TASKPOLL INIT statement.

Use of the TASKREC function is illustrated in the *HandleInGoAway* subroutine. When a user clicks the mouse in the *close box* of a window, the window is hidden. The window can be made visible again by selecting its name from the Windows menu. When TASKPOLL detects an *InGoAway* event, the *HandleInGoAway* subroutine is called. In order for the *HandleInGoAway* subroutine to determine which window to close, it must examine the TaskData field of the Task Record. For example:

```
HandleInGoAway:
   theWindow@ = TASKREC@(8)
   _HideWindow(theWindow@)
   RETURN 0
```

The main event loop may contain other operations which maintain the current state of the desktop. For example, the GSDEMO.BAS program, enables and disables the Edit menu items depending upon the type of the topmost window. The GSDEMO.BAS windows do not support the editing operations found in the Edit menu. Therefore, they should be disabled in order to communicate to the user that they have no affect. However, if a desk accessory window is topmost, the editing operations may be supported, and thus enabled. The main event loop contains a call to the procedure *CheckMenus* which checks the topmost window and enables or disables the Edit menu items appropriately. Most desktop applications should implement a *CheckMenus* procedure, and depending upon the nature of the application, other operations may be appropriate as well.

Summary

This chapter has introduced the principles of well-engineered desktop applications which follow the Apple Human Interface Guidelines. In addition, the techniques for writing event-driven programs which use the Apple IIGS Toolbox have been reviewed in the context of the GSDEMO.BAS application.

Other desktop applications can be found in the MORE.EXAMPLES folder. These examples further illustrate the techniques for writing event-driven programs.

Part IV

Appendices

Appendix A
Error Messages

This appendix lists all editor, compiler, linker and runtime errors which may occur while using TML BASIC. All errors are reported in the standard Error Dialog Box as described in Chapter 3. Each error is also displayed with an icon which indicates the component of TML BASIC which detected the error. Editor errors are displayed with the upside down yield sign, compiler errors with a green bug, linker errors with two chain links, and runtime errors with an exploding bomb.

Explanatory notes follow most of the error messages to help clarify their meaning, and in some cases additional notes appear explaining how to correct the error.

Some messages contain the caret character (^) which is substituted by TML BASIC at the time the error is displayed with an identifier, label, or some other value to help make the error message more meaningful. For example, the compiler error message:

> Procedure "^" is not declared

might appear in the error dialog as:

> Procedure "DrawBoxes" is not declared

if TML BASIC detected that the procedure DrawBoxes was called but not declared.

TML BASIC Editor Errors

Memory is getting low. Close a document window.

> TML BASIC has detected that you are running dangerously low on memory. In order to avoid the potential loss of data, you are recommended to free memory by closing a document window. You may also choose the *Compact Memory* option from the Preferences dialog. See Chapter 6.

Error reading file.

> An error occurred while reading the document from disk. This might happen if the file is damaged or the disk has been removed from the disk drive.

Error saving file.

This error is reported when TML BASIC is unable to save the contents of a document window to disk. This is usually occurs because the disk is locked, removed from the disk drive or the disk is full.

Error deleting file.

This error is reported after you have chosen to delete a disk file using the **Delete...** command from the **ProDOS** menu and the disk has been removed from the disk drive or the disk is locked.

Error renaming file.

This error is reported after you have chosen to rename a disk file using the **Rename...** command from the **ProDOS** menu and you have specified an illegal filename, the disk has been removed from the disk drive or the disk is locked.

Error occurred while loading Print Tools.

This error only occurs in the Network version of TML BASIC. The message indicates that one or more of the necessary system files needed to use the Print Manager could not be loaded to memory.

Can't open that file. File already open in another window.

You are not allowed to open the same file more than once.

Unable to complete that operation. File would become too large.

The editing operation you just attempted would have caused the file to become greater than 32K bytes.

File too large to open. Maximum file size is 32K bytes.

TML BASIC can only open files which are less than or equal to 32K bytes in size.

Insufficient memory available to open that file.

> Even though the file you are attempting to open is smaller than 32K bytes, there is insufficient memory available to read the file into memory.

Insufficient memory to complete that operation.

> An operation has failed due to the lack of available memory.

TML BASIC Compiler Errors

Lexical Errors

String constant must not exceed source line.

> A string constant literal is missing its closing quote.

Error in numeric literal.

> The syntax for a numeric literal value is incorrect.

Illegal character in input.

> An illegal character has been detected in the source file. See Chapter 7 for the legal BASIC characters.

Syntax Errors

Identifier expected.
String constant expected.
Integer constant expected.
")" expected.
"(" expected.
":" expected.
"," expected.
";" expected.
"THEN" expected.
"END PROC" or "END FN" expected.
"END LIBRARY" expected.
Unexpected symbol.

> These error messages indicate that the program contains illegal BASIC syntax. While the error message indicates the symbol expected at the time the error was detected, it is

possible that other symbols could also repair the syntax error. If you are unfamiliar with BASIC syntax then you should study Chapters 7 through 10.

Error in Expression.

An expression containing illegal BASIC syntax was detected that contains an error for which there is no specific error message.

Error in Statement.

A statement containing illegal BASIC syntax was detected that contains an error for which there is no specific error message.

THEN without matching IF
ELSE without matching IF

The THEN or ELSE statement appeared on a line by itself without a preceding IF statement.

Block IF statement without matching END IF
ELSEIF / END IF statement without matching Block IF statement.

The Block IF, ELSEIF or END IF statements appear without the necessary matching statements. For further information see Chapter 10.

UNTIL without matching DO or WHILE.
WHILE without matching UNTIL.
DO without matching UNTIL.

The DO, WHILE or UNTIL statements appear without the necessary matching statements. For further information see Chapter 10.

NEXT without matching FOR.
FOR without matching NEXT.

The FOR or NEXT statements appear without the necessary matching statement. For further information see Chapter 10.

Semantic Errors

Type Mismatch Error.

The type of an expression is inappropriate for the context in which it is used. For example, assigning a string expression to a numeric variable or vice versa.

May not declare LOCAL arrays.

The LOCAL statement can only be used to declare simple variables.

Duplicate declaration of LOCAL variable or parameter.

A LOCAL statement attempted to declare the same name as previously declared in a LOCAL statement or the same name as a parameter.

Illegal use of the LOCAL statement.

The program attempted to use the LOCAL statement in the main program or in a procedure or function, but after an executable statement.

Illegal Parameter Type.

It is not legal to have a structure array element as a parameter type.

Procedure "^" is not declared.

The user defined procedure or function referenced in a PROC or FN statement is not defined in the program.

Toolbox procedure "^" is not defined.

The Toolbox procedure or function referenced in a CALL or EXFN_ statement is not defined. Make sure that the name is spelled correctly and that the required Toolbox library name appears in a LIBRARY statement.

Number of parameters does not match declaration.

The number of parameters in the parameter list of a procedure, function or toolbox routine call does not properly match its declaration. Check the declaration of the procedure, function or toolbox routine.

Label "^" is referenced, but not defined.

The specified label is referenced in a GOTO, GOSUB, EVENTDEF, or MENUDEF statement; but is not defined anywhere in the program.

Duplicate declaration of a static array.

An array variable can only appear once in a DIM statement throughout the entire program. If an array must be redimensioned in a program, the DIM DYNAMIC statement must be used. See Chapter 7.

Static arrays must have constant dimensions.

An array variable in a DIM statement must be dimensioned with static values. If an array must be dynamically dimensioned, the DIM DYNAMIC statement must be used. See Chapter 7.

Arrays limited to eight dimensions.

The maximum number of dimensions an array may have is eight.

Library Errors

Unable to find/open library file.

The ".LIB" file for the named library cannot be found in either the *current unit prefix* or the *Library File search directory* specified in the Preferences dialog.

Unable to write compiled library file.

The compiler is unable to create, open or write to the library's ".LIB" file. The disk may be locked, removed from the disk drive or full.

Incompatible version of library file.

> Whenever you receive a new version of TML BASIC you must recompile all of your libraries.

Symbol table space exhausted.

> The number of declarations in this library has exhausted the available memory allocated for the library's symbol table. You should adjust the symbol table size in the Preferences dialog. See Chapter 6.

TML BASIC Linker Errors

Out of Memory.

> Insufficient memory is available for the linker to allocate the data structures it requires to link the program. Try closing a document window to release memory.

Segment "^" specified as both CODE and DATA.

> You have specified the same segment name in a {*$CodeSegment segname*} and {*$DataSegment segname*} compiler directive.

Segment "^" too large.

> A CODE or DATA segment became larger than 64K bytes. You must resegment your program so that the segment does not exceed this limit. See Appendix B.

Unresolved linker reference to symbol "^".

> An externally defined label cannot be found by the linker. You should recheck the spelling of the symbol to make sure it is correct.

Unable to create/open application file.

> After a Compile To Disk completes successfully, the linker attempts to write the application file to disk. The error is reported if this file cannot be created and/or opened. This usually happens when the disk is locked or has been

removed from the disk drive.

Error in writing to application file.

This error is reported when TML BASIC was able to create and/or open the output application file, but encountered an error during writing. This is usually caused because of a locked disk or a disk becoming too full to write the entire file contents of the file to disk.

TML BASIC Runtime Errors

Runtime errors are detected during program execution. That is, the compiler has successfully compiled the program without any lexical, syntax or semantic errors and has generated machine code for the program. However, when the program runs, an error occurs. If a program is run directly from TML BASIC using the **To Memory & Run** compile option, TML BASIC selects the line of text containing the runtime error and displays the error message in the standard Error Dialog Box. If the error is detected in a compiled to disk application, execution aborts and displays the runtime error number. The error number should be compared to those below to determine the error which occurred.

Following is the list of possible runtime errors:

1 Overflow Error.

This error occurs when the result of a numeric calculation produces a value which is too large (or too small) to be represented in the indicated numeric type.

2 Illegal Quantity Error.

A parameter to a function or statement was not in the legal range of values specified for the function or statement.

3 Out of DATA.
A READ statement ran out of DATA statement values.

4 Divide by 0.

An attempt was made to divide by 0 or raise zero to a negative power.

5 RETURN/POP without matching GOSUB.

A RETURN or POP statement was executed without a matching GOSUB. That is, there is nothing to return to.

6 Program Interrupted.

The STOP statement was executed or the program was aborted by typing the Control-C character.

7 Out of Memory.

Many situations may cause this error to arise. If your program attempts to create a dynamic array using the DIM DYNAMIC statement for which there is insufficient memory to allocate, the error occurs. Or if the string pool overflows.

8 RESUME without an error.

The program executed a RESUME statement while not handling an error.

9 File Not Open.

The program attempted to use a file reference number which has not been associated with a file via the OPEN statement.

10 File Open.

This error occurs if a program attempts to open, rename or delete a file which is currently open.

11 File (Path) Not Found.

The pathname specified in a TML BASIC I/O statement or function is either illegal or the file does not exist.

12 Volume Not Found.

The volume name specified in a pathname of an I/O statement does not match the volume name of any currently mounted volume.

13 File Locked.

The program attempted to modify a locked file.

14 File Type Error.

This error occurs when the program attempts to reference a file in such a way which is incompatible with its file type. See Chapter 10 I/O statements.

15 Duplicate File Error.

This error occurs when the CREATE statement is used to create an already existing file.

16 Write Protect Error.

The program attempted to modify a file on a write-protected disk.

17 Device Not Found.

This error occurs when the program specifies a device name which contains an illegal character.

18 Bad Path Error

This error occurs whenever an illegal character appears in a pathname.

19 Disk Full.

This error is reported when there is no additional space left on the disk necessary to complete an I/O statement.

20 Illegal Using Specification

The using specification or IMAGE statement contained an illegal specification definition.

21 Stack Overflow Error.

The program contained too many procedure, function or subroutine calls; or declared too many local variables for the size of the *Runtime Stack*. This error is only detected if the Check Stack option is turned on in the Preferences Dialog.

Appendix B
Metastatements

Metastatements are compiler directives. Strictly speaking, metastatements are not part of the TML BASIC language, but rather a special construct used by programs to control the behavior of the TML BASIC compiler. For example, a metastatement can be used to specify the size of the string pool.

Most of the metastatements correspond to an option in the Preferences Dialog. However, the effect of a metastatement *always* overrides the setting of an option in the Preferences Dialog. The metastatements which also appear in the Preferences Dialog are:

$CheckStack
$Debug
$EventTrapping
$KeyboardBreak
$OnError
$StackSize
$StringPoolSize

See Chapter 6 for information regarding the Preferences Dialog.

A metastatement must appear on a line by itself, and must begin with a dollar sign ($), immediately followed by the name of the metastatement. If the name does not spell one of the legal TML BASIC metastatements, TML BASIC reports the error "Unknown metastatement". Following the name are the metastatement arguments. An argument can be the reserved words ON and OFF, a numeric constant, or a string constant. The following code fragment shows how metastatements can be used in a program.

```
$StringPoolSize 1
$KeyboardBreak OFF
$Debug ON
INPUT "How many numbers to average: ", howMany%
Total = 0
FOR i% = 1 to howMany%
   PRINT "Number "; i%; ": "
   INPUT nextValue
   Total = Total + nextValue
NEXT i%
PRINT "The average is: "; Total/howMany%
```

The following paragraphs describe the actions and arguments for each of the TML BASIC metastatements.

$CheckStack

Syntax: $CheckStack ON | OFF

Default: $CheckStack OFF

As discussed in Chapter 8, TML BASIC implements a data structure called the *Runtime Stack*. This stack is used for implementing the GOSUB, PROC and FN statements as well as allocating storage for parameters and local variables with the LOCAL statement. The stack has a limited fixed size as specified by $StackSize metastatement or its corresponding option in the Preferences Dialog. Thus, it is possible to write programs which exceed the storage capacity of the stack.

This option is used to instruct the TML BASIC compiler to generate special code for each procedure and function's entry code. This code checks to be sure that there is sufficient space in the runtime stack to call the procedure and allocate its local variables. If there is insufficient space, the runtime error "Stack Overflow" occurs. If the Debug option is turned on then the TML BASIC debugger is capable of showing you what procedure or function caused the stack overflow.

Most programs never need more stack space than the default 8K bytes, thus this option is turned off by default. However, if your program is behaving very strangely, it may be that its stack is growing too large and destroying memory. Turning this option on will determine if your program does indeed have this problem. If it does, you should increase the allocated stack space for the program.

$CodeSegment

Syntax: $CodeSegment *"segmentname"*

Default: $CodeSegment "main"

An Apple IIGS application may consist of one or more *code segments.* A code segment is a special Apple IIGS data structure implemented by the System Loader. A code segment contains the binary code for a program, its relocation information and other special values. Every code segment has a unique name.

TML BASIC programs have two code segments by default: "main" and "mainprog". The segment "mainprog" contains all the code for the main program, while the segment "main" contains all the code for user defined procedures and functions. In addition, the code for separately compiled libraries is contained in the code segment "main". Small programs usually only consist of these two segments, but larger programs must be divided into several code segments because the Apple IIGS limits the size of an individual code segment to 64K bytes. The reason for the size restriction is that a code segment must not cross the boundries of a *bank* of memory. On the Apple IIGS, a bank of memory is 64K bytes.

The $CodeSegment metastatement is used to control the code segment that code for procedures and functions are generated. It is not possible to segment the main program (statements not contained in a procedure or function). When the $CodeSegment metastatement appears in the source code, the code for all subsequent procedures and functions is generated to the new code segment.

$DataSegment

Syntax: $DataSegment *"segmentname"*

Default: $DataSegment "~global"

Like code segments, an Apple IIGS application may consist of one or more *data segments*. A data segment is a special Apple IIGS data structure implemented by the System Loader. A data segment contains the information necessary to allocate storage and initialize memory for global variables. Every data segment has a unique name.

TML BASIC programs have one data segment by default: "~global". Programs usually only consist of a single data segment, but programs which declare a large number of global variables or arrays which are very large, must divide the global storage into several data segments because the Apple IIGS limits the size of an individual data segment to 64K bytes. The reason for the size restriction is that a data segment must not cross the boundries of a *bank* of memory. On the Aple IIGS, a bank of memory is 64K bytes.

The $DataSegment metastatement is used to control the data segment in which storage for global variables are allocated. TML BASIC allocates storage for a global variable or array when it is first used, or when an array variable is declared in a DIM statement. At the point of declaration, TML BASIC allocates storage for the variable in the current data segment. To change the current data segment use the $DataSegment metastatement.

$Debug

Syntax: $Debug ON | OFF

Default: $Debug ON

The $Debug metastatement is used to control the generation of debugging code by the TML BASIC compiler which is used by the TML BASIC debugger to detect and report runtime errors. The generated code checks for all runtime errors, such as Overflow Error, Illegal Quantity Error, etc. It also generates a special data structure called the *line number table* so that the TML BASIC debugger can determine in what line of source code the runtime occurred. If this option is turned off, all runtime errors will go undetected. The runtime errors are listed in Appendix A.

The Debug option makes programs larger and slower to execute. The option should be turned off when a program is known to be correct, and no longer requires the debug code.

$EventTrapping

Syntax: $EventTrapping ON | OFF

Default: $EventTrapping ON

This metastatement must be turned on when a program contains statements requiring event trapping. These statements are the ON KBD and ON TIMER. When these statements are used, TML BASIC must generate code between each statement to check for the occurrence of a keyboard or timer event. This option should only be turned on when a program contains these statements since the code necessary to check for these events makes a program larger and slower to execute.

If a program contains the statements ON KBD or ON TIMER, and event trapping is turned off, TML BASIC will report an error. See Chapter 10 for more information regarding these two statements.

$KeyboardBreak

Syntax: $KeyboardBreak ON | OFF

Default: $KeyboardBreak ON

The KeyboardBreak metastatement is used to implement the ON BREAK statement. It is also required to allow a program to be aborted by typing a control-C.

If this option is turned on, TML BASIC generates code between each statement to check if the control-C character has been typed. If this option is turned off, it is impossible to abort the execution of a TML BASIC program. The only way to do so is to reset the Apple IIGS. If you do not intend to abort the execution of your programs and you do not use the ON BREAK statement, then you should turn this option off so that your programs will be smaller and, in turn, run faster.

$OnError

Syntax: $OnError ON | OFF

Default: $OnError ON

The OnError metastatement is used to indicate to the TML BASIC compiler that the program contains the ON ERR statement along with the statements RESUME and/or RESUME NEXT. These statements require that the *line number table* be generated so that TML BASIC can determine on which line to resume or resume next after an error has been handled by an ON ERR statement list.

If your programs do not contain these statements then it is best to turn this option off since it will decrease the size of your applications. However, if this option is turned off and a program contains the ON ERR, RESUME or RESUME NEXT statements, TML BASIC will report an error.

$StackSize

Syntax: $StackSize *number*

Default: $StackSize 8

As described in the $CheckStack metastatement, TML BASIC implements a data structure called the *Runtime Stack* for implementing the GOSUB, PROC, FN and LOCAL statements. The default size for the Runtime Stack is 8K bytes. However, some programs may require a larger stack because it uses a large number of procedure calls or local variables. The stack size can be changed with the $StackSize metastatement to a size from 1K to 32K bytes. The argument to the $StackSize metastatement must be a numeric integer constant. The value is expressed in K-bytes. Thus, the value 1 means 1K bytes, 2 means 2K bytes, etc. See Chapter 8 for more information about the Runtime Stack.

$StringPoolSize

Syntax: $StringPoolSize *number*

Default: $StringPoolSize 10

The values for all string variables and string constants are stored in a special data structure called the *String Pool*. The string pool has a fixed, limited size which is set by the $StringPoolSize metastatement or its corresponding option in the Preferences Dialog. The default size of the string pool is 10K bytes. If a program is running out of string space this value should be increased. The maximum size for the string pool is 64K bytes, the minimum is 1K bytes. The value is expressed in K-bytes. Thus, the value 1 means 1K bytes, 2 means 2K bytes, etc. For more information about strings, string data and the string pool see Chapter 7.

Appendix C

Apple IIGS Toolbox Libraries

As discussed in Chapters 11 through 13 of this manual, the Apple IIGS Toolbox is the large collection of software routines developed by Apple Computer and built into every Apple IIGS computer. The Toolbox routines implement the super hi-res graphics screen and the QuickDraw graphics engine. It also implements sound, menus, windows, dialogs and much more. As discussed in Chapter 11, the Toolbox is divided into a collection of *tool sets* (or *managers*). Each of these tool sets implements a related collection of procedures, functions and data structures.

TML BASIC provides programmer access to the Toolbox with a collection of libraries, each defining the interface to an individual tool set. The libraries are shipped on the TML BASIC distribution disk in the LIBRARIES folder. The source code to the libraries is not provided, however, the contents of each library is listed in this Appendix in alphabetical order. The libraries provided with TML BASIC are shown in the following table.

Apple IIGS Tool Set	**TML BASIC Library Name**
Control Manager	Control
Desk Manager	Desk
Dialog Manager	Dialog
Event Manager	Event
Font Manager	Font
Integer Math	IntMath
Line Edit	LineEdit
List Manager	List
Memory Manager	Memory
Menu Manager	Menu
Miscellaneous Tools	MiscTool
Note Synthesizer	NoteSyn
Print Manager	Print
QuickDraw	QuickDraw
QuickDraw Auxiliary	QDAux
Scheduler	Scheduler
Scrap Manager	Scrap
Sound Manager	Sound
Standard File	StdFile
Text Tools	TextTool
Tool Locator	ToolLocator
Window Manager	Window

The discussion of each tool set is divided into four parts: Introduction, Special Values, Data Structures and Routines. The following paragraphs describe the contents of these parts and the notational conventions used. While this appendix provides a thorough description of the *contents* of each tool set, and the data structures they use, it is in no way a substitute for a good reference on *how* the Toolbox works. The definitive reference is of course Apple's technical publication entitled *Apple IIGS Toolbox Reference: Volumes 1 and 2*. Anyone attempting to program the Toolbox beyond the simple use of QuickDraw graphics which is fully documented in Chapter 12 should obtain a good reference describing the details of the Toolbox.

Introduction

Before describing the data structures, procedures and functions of a tool set, a brief description of the particular tool set's functions and capabilities is given.

Special Values

This section is used to highlight various important values used or returned by the procedures and functions in the tool set. For example, the Memory Manager provides the function NewHandle@ which is used to allocate a block of memory. One of the parameters to this function specifies various attributes about the block to be allocated. The attributes are defined by the program with special predefined values. If you did not know the meaning of the values for this parameter, it would be impossible to properly allocate a block of memory.

Not every conceivable special value used by the procedures and fucncions in a tool set is defined in this section, but only the most important and most commonly used values. The special values are presented in a table which defines the integer value and its meaning. In some cases, a paragraph is provided which describes how the collection of values are used.

Data Structures

Many of the Toolbox procedures and functions manipulate *data structures* rather than just simple values. Data structures are collections of values grouped together into a single variable. In TML BASIC these data structures are represented as array or structure variables. Because BASIC does not offer a *typing* mechanism similar to those found in languages like Pascal or C, it is not possible to define new types from which variables can be declared. Instead, the programmer must declare array and/or structure variables in an appropriate fashion and use them in a way consistent with the meaning the Toolbox has given a particular data structure.

This section of the appendix provides a *template* of how an array or structure variable might be declared and used for a particular Toolbox data structure. The template includes a paragraph describing the purpose of the data structure, an example DIM statement that a program might use to create an instance of the data structure, and a definition of the meaning for each element in the array or structure variable. The example template is not necessarily the only way the data structure might be defined, but is generally the best.

The following is an example of the *Point* data structure used by the QuickDraw graphics routines. A Point defines a location in the two-dimensional drawing space of the QuickDraw super hi-res screen. As such, it includes two integer values which define the horizontal and vertical position of the Point.

```
DIM aPoint%(1)
```

Element(s)	**Value**	**Description**
0	Integer	Horizontal coordinate
1	Integer	Vertical coordinate

As seen in the data structure definition, a Point has been described as an integer array containing two elements. The first element is the horizontal coordinate and the second element is the vertical coordinate.

Data structures which do not contain elements all of the same type are usually defined as a structure rather than an array. Certain ranges of bytes within the structure then make up each element of the data structure. In this case, simple assignments to each element of the data structure are not possible. Instead, the SET and VAL statements are used to access the elements of the data structure. The following is the *PenState* data structure also used by QuickDraw.

```
DIM aPenState!(47)
```

Element(s)	**Value**	**Description**
0..3	*Point*	Pen location
4..7	*Point*	Pen size
8..39	*Pattern*	Pen pattern
40..47	*Mask*	Pen mask

In this example, we see that the first two elements of the data structure are *Point*s. A Point is another data structure which we declared above. A Point is an array of two integers, thus the Point elements occupy four bytes each in the PenState data structure. The Pattern and Mask are other data structures defined by QuickDraw.

Routines

The Toolbox is not part of the TML BASIC language, but an integral part of the Apple IIGS. The code which implements the Toolbox routines is written in assembly language and is stored in the Apple IIGS ROM (read-only memory) or on the System Disk and read into RAM (random access memory). As such, the procedures and functions in each tool set are not defined in the normal fashion using the DEF PROC and DEF FN statements, nor are they called in the normal way using the PROC or FN statements. As discussed in Chapter 11, TML BASIC provides the CALL and EXFN statements for calling the Toolbox procedures and functions. However, no mechanism is provided in TML BASIC for ***defining*** a Toolbox procedure or function.

Since the Toolbox procedures and functions cannot be defined using legal TML BASIC statements, the Toolbox libraries are created with a special tool not shipped on the TML BASIC distribution disk. However, to define each Toolbox routine name and its parameters we have adopted a special notation based upon the familiar DEF PROC and DEF FN statements for this appendix.

Each declaration begins with either DEF PROC or DEF FN depending if the routine is a procedure or a function. Following this is the name of the Toolbox routine. These names match exactly those documented in the *Apple IIGS Toolbox Reference*. If the routine is a function then the the name is followed by a type character to indicate the result type of the function. Unlike normal BASIC functions, a Toolbox function might return more than one value (using R.STACK, see Chapter 11). In this case, square brackets are used to indicate the number of function return values. After this is the list of parameters enclosed in parenthesis. The declaration ends with the word TOOL followed by two integers separated by a comma. The word TOOL indicates that the declaration is a Toolbox routine rather than a normal procedure or function that would contain code and end with a END PROC or END FN. The following is the sytnax of a Toolbox procedure and function declaration:

```
DEF PROC ToolName[ ( Parameter {, Parameter} ) ]  TOOL FunctionNum,ToolNum

DEF FN ToolName[!|%|@|&] [ [NumReturnValues] ]
   [ ( Parameter {, Parameter} ) ]  TOOL FunctionNum,ToolNum
```

A Toolbox routine can be invoked using the CALL statement followed by the name of the routine as defined in this appendix, or using the CALL% statement using the routine's FunctionNum and ToolNum.

The parameter names chosen for this appendix are intended to describe the meaning of the parameter's value. For example, a parameter with the name UserID%, indicates that a Memory Manager user id should be passed for the parameter value. The type character following the name of course indicates the type of the parameter. In this case, UserID% is an integer parameter.

Often times, a data structure value is passed to a Toolbox routine. In these cases, the name of the parameter includes as part of its name, the name of the data structure in italics. For example, the Toolbox function PtinRect% contains the parameter *Point*Ptr@. Since TML BASIC does not allow array and structure parameters, the address of the data structure must be passed as a double integer (@) parameter. The address of a array or structure variable is obtained by using the VARPTR function. Whenever the address of a data structure is used as a parameter, its name usually ends with the letters "Ptr". Toolbox routines sometimes require *Handle* values for a parameter. A handle is a pointer to a pointer. While handles are used extensively in the Toolbox, only in rare occasions does a program have to create a handle value using VARPTR since the Toolbox itself creates them and returns the handles to the program. When a parameter value is a handle, its name ends with the letters "Hndl".

Control Manager

The Control Manager consists of all the routines which manipulate controls. Controls include scroll bars, radio buttons, check boxes, etc. When a control is activated or selected it causes an immediate action to take place or changes a setting that affects the operation of the application or the window to which the control belongs.

Special Values

No special values defined for the Control Manager.

Data Structures

ControlColorTbl

The ControlColorTbl structure holds the information used to add color to a control. Because the parts of a control differ among the different controls, the contents of this structure vary according to the type of control used. The following is the definition of the most common form of the ControlColorTbl, for a scroll bar. The bits in each of the integer elements of the data structure define the colors. For further details, consult the *Apple IIGS Toolbox Reference.*

```
DIM aControlColorTbl%(7)
```

Element	Value	Definition
0	Integer	Scroll outline color
1	Integer	Color of arrows when not highlighted
2	Integer	Color of arrows when highlighted
3	Integer	Color of arrow box interior background
4	Integer	Color of thumb interior when not highlighted
5	Integer	Reserved
6	Integer	Color of page region interior
7	Integer	Color of scroll bar interior when inactive

Control

The Control data structure defines the actual control object. Like the ControlColorTbl data structure, the exact elements of this data structure depends on the control being defined. The control listed here is for a scroll bar. For further details see the *Apple IIGS Toolbox Reference.*

```
DIM Control!(46)
```

Element	Value	Definition
0..3	Double Integer	Handle to the next control in the control list
3..6	Double Integer	Pointer to the window that owns this control: *Window*Ptr
7..14	*Rect*	Control rectangle
15	Integer	Control flag
16	Integer	Control hilite
17..18	Integer	Value of the control
19..22	Double Integer	Pointer to control definition procedure: *Proc*Ptr
23..26	Double Integer	Pointer to control default action procedure: *Proc*Ptr
27..28	Integer	Data size
29..30	Integer	View size
31..34	Double Integer	Reserved for application's use
35..38	Double Integer	Pointer to control color table: *ControlColorTbl*Ptr
39..42	*Rect*	Thumb rectangle
43..46	*Rect*	Page rectangle

Routines

HouseKeeping

```
DEF PROC   CtlStartUp(UserID%,DPageAddr%)                   TOOL 2,16
DEF PROC   CtlShutDown                                      TOOL 3,16
DEF FN     CtlVersion%                                      TOOL 4,16
DEF PROC   CtlReset                                         TOOL 5,16
DEF FN     CtlStatus%                                       TOOL 6,16
DEF PROC   CtlNewRes                                        TOOL 18,16
```

Creating and Disposing

```
DEF FN     NewControl@
               (TheWindowPtr@,
                BoundsRectPtr@,
                TitleString$,
                Flag%,
                Value%,
                Param1%,
                Param2%,
                DefProcPtr@,
                RefCon@,
                ControlColorTablePtr@)                            TOOL 9,16
DEF PROC   DisposeControl(TheControlHndl@)                        TOOL 10,16
DEF PROC   KillControls(TheWindowPtr@)                            TOOL 11,16
```

Display

```
DEF PROC   SetCtlTitle(TitleString$,TheControlHndl@)              TOOL 12,16
DEF FN     GetCtlTitle@(TheControlHndl@)                          TOOL 13,16
DEF PROC   HideControl(TheControlHndl@)                           TOOL 14,16
DEF PROC   ShowControl(TheControlHndl@)                           TOOL 15,16
DEF PROC   DrawControls(TheWindowPtr@)                            TOOL 16,16
DEF PROC   HiliteControl(HiliteState%,TheControlHndl@)            TOOL 17,16
```

Mouse Location

```
DEF FN     FindControl%
               (FoundControlHndl@,
                XPoint%,
                YPoint%,
                TheWindowPtr@)                                    TOOL 19,16
DEF FN     TestControl%
               (XPoint%,
                YPoint%,
                TheControlHndl@)                                  TOOL 20,16
DEF FN     TrackControl%
               (StartX%,
                StartY%,
                ActionProcPtr@,
                TheControlHndl@)                                  TOOL 21,16
```

Moving and Sizing

```
DEF PROC   MoveControl(NewX%,NewY%,TheControlHndl@)               TOOL 22,16
DEF PROC   DragControl
               (StartX%,
                StartY%,
                LimitRectPtr@,
                SlopRectPtr@,
                Axis%,
                TheControlHndl@)                                  TOOL 23,16
```

Control Record Access

```
DEF PROC  SetCtlValue(CurValue%,TheControlHndl@)                TOOL 25,16
DEF FN    GetCtlValue%(TheControlHndl@)                         TOOL 26,16
DEF PROC  SetCtlAction(NewActionProcPtr@,TheControlHndl@)       TOOL 32,16
DEF FN    GetCtlAction@(TheControlHndl@)                        TOOL 33,16
DEF PROC  SetCtlRefCon(NewRefCon@,TheControlHndl@)              TOOL 34,16
DEF FN    GetCtlRefCon@(TheControlHandle@)                      TOOL 35,16
DEF PROC  SetCtlParams(Param2%,Param1%,TheControlHndl@)         TOOL 27,16
DEF FN    GetCtlParams%[2](TheControlHndl@)                     TOOL 28,16
```

Miscellaneous

```
DEF FN    DragRect@
              (ActionProcPtr@,
               DragPatternPtr@,
               StartX%,
               StartY%,
               DragRectPtr@,
               LimitRectPtr@,
               SlopRectPtr@,
               Axis%)                                           TOOL 29,16

DEF FN    GetCtlDPage%                                          TOOL 31,16
DEF FN    GrowSize@                                             TOOL 30,16
DEF FN    SetCtlIcons@(NewFontHndl@)                            TOOL 24,16
DEF PROC  EraseControl(TheControlHndl@)                         TOOL 36,16
DEF PROC  DrawOneCtl(TheControlHndl@)                           TOOL 37,16
```

Desk Manager

The Desk Manager is the tool set that allows applications to support Classic Desk Accessories and New Desk Accessories. Classic Desk Accessories (CDA) are invoked with a keyboard interrupt generated by the Open Apple-Control-Escape key sequence. New Desk Accessories (NDA) can only be invoked by applications that support the desktop environment. NDAs are usually available in the Apple menu of an application.

Special Values

The following values are the codes passed to a DAActive procedure.

Value	Definition
1	Desk Accessory Event
2	Desk Accessory Run
3	Desk Accessory Cursor
4	Desk Accessory Menu
5	Desk Accessory Undo
6	Desk Accessory Cut
7	Desk Accessory Copy
8	Desk Accessory Paste
9	Desk Accessory Clear

Data Structures

No data structures defined for the Desk Manager.

Routines

HouseKeeping

```
DEF PROC  DeskStartUp                                    TOOL 2,5
DEF PROC  DeskShutDown                                   TOOL 3,5
DEF FN    DeskVersion%                                   TOOL 4,5
DEF PROC  DeskReset                                      TOOL 5,5
DEF FN    DeskStatus%                                    TOOL 6,5
```

State Save and Restore

```
DEF PROC  SaveScrn                                       TOOL 9,5
DEF PROC  RestScrn                                       TOOL 10,5
DEF PROC  SaveAll                                        TOOL 11,5
DEF PROC  RestAll                                        TOOL 12,5
```

Installation

```
DEF PROC   InstallNDA(IDHndl@)                          TOOL 14,5
DEF PROC   InstallCDA(IDHndl@)                          TOOL 15,5
```

Classic Desk Accessory

```
DEF PROC   ChooseCDA                                    TOOL 17,5
DEF PROC   SetDAStrPtr(AltDispHndl@,StringTblPtr@)      TOOL 19,5
DEF FN     GetDAStrPtr@(DAIDNum%)                       TOOL 20,5
```

New Desk Accessory

```
DEF FN     OpenNDA%(IDNum%)                             TOOL 21,5
DEF PROC   CloseNDA(RefNum%)                            TOOL 22,5
DEF PROC   CloseNDAbyWinPtr(TheWindowPtr@)              TOOL 28,5
DEF PROC   CloseAllNDAs                                 TOOL 29,5
DEF PROC   FixAppleMenu(MenuNum%)                       TOOL 30,5
DEF FN     GetNumNDAs%                                  TOOL 27,5
DEF PROC   SystemClick
               (TheEventRecord@,
                TheWindowPtr@,
                Flags%)                                 TOOL 23,5
DEF FN     SystemEdit%(EditType%)                       TOOL 24,5
DEF PROC   SystemTask                                   TOOL 25,5
DEF FN     SystemEvent%
               (What%,
                Message@,
                When@,
                WherePointPtr@,
                Mods%)                                  TOOL 26,5
```

Dialog Manager

The Dialog Manager contains routines for manipulating dialog and alert boxes. These boxes provide clear consistent ways for the application to communicate with the user. Dialog boxes are used primarily to request certain types of input while alert boxes warn the user of an impending situation.

Special Values

The first two values listed are the standard dialog item id numbers for the OK and Cancel buttons. The remaining values are used when creating a new dialog item to define the type of dialog item to create.

Value	Definition
1	Ok
2	Cancel
10	Button Item
11	Check Item
12	Radio Item
13	ScrollBar Item
14	User Control Item
15	StaticText Item
16	Long StaticText Item
17	Edit Line Item
18	Icon Item
19	Picture Item
20	User Item

Data Structures

Dialog

The Dialog data structure is a complex variable size structure containing pointers and handles to other structures in memory. The contents of the Dialog data structure are not public. Instead, to manipulate dialogs and their contents, the programmer uses the standard routines found in the Dialog Manager.

Routines

HouseKeeping

```
DEF PROC  DialogStartUp(UserID%)                      TOOL 2,21
DEF PROC  DialogShutDown                              TOOL 3,21
DEF FN    DialogVersion%                              TOOL 4,21
DEF PROC  DialogReset                                 TOOL 5,21
DEF FN    DialogStatus%                               TOOL 6,21

DEF PROC  ErrorSound(SoundProcPtr@)                   TOOL 9,21
DEF PROC  SetDAFont(FontHndl@)                        TOOL 28,21
```

Creating and Disposing

```
DEF FN    NewModalDialog@
              (dBoundsRectPtr@,
               dVisible%,
               dRefCon@)                              TOOL 10,21
DEF FN    NewModelessDialog@
              (dBoundsRectPtr@,
               dTitleString$,
               dBehindWindowPtr@,
               dFlag%,
               dRefCon@,
               dFullSizeRectPtr@)                     TOOL 11,21
DEF FN    GetNewModalDialog@(TheDialogPtr@)           TOOL 50,21
DEF PROC  CloseDialog(TheDialogPtr@)                  TOOL 12,21
```

Creating and Removing Items

```
DEF PROC  NewDItem
                (TheDialogPtr@,
                 ItemID%,
                 ItemRectPtr@,
                 ItemType%,
                 ItemDescrUNIVPtr@,
                 ItemValue%,
                 ItemFlag%,
                 ItemColorTablePtr@)                  TOOL 13,21
DEF PROC  GetNewDItem(TheDialogPtr@,ItemTemplatePtr@) TOOL 51,21
DEF PROC  RemoveDItem(TheDialogPtr@,ItemID%)          TOOL 14,21
```

Handling Dialog Events

```
DEF FN    ModalDialog%(FilterProcPtr@)                TOOL 15,21
DEF FN    ModalDialog2@(FilterProcPtr@)               TOOL 44,21
DEF FN    IsDialogEvent%(TheEventRecordPtr@)          TOOL 16,21
DEF FN    DialogSelect%
                (TheEventRecordPtr@,
                 TheDialogPtr@,
                 ItemHit%)                            TOOL 17,21
DEF PROC  DlgCut(TheDialogPtr@)                       TOOL 18,21
DEF PROC  DlgCopy(TheDialogPtr@)                            TOOL 19,21
DEF PROC  DlgPaste(TheDialogPtr@)                           TOOL 20,21
DEF PROC  DlgDelete(TheDialogPtr@)                          TOOL 21,21
DEF PROC  DrawDialog(TheDialogPtr@)                         TOOL 22,21
```

Invoking Alerts

```
DEF FN    Alert%(AlertTemplatePtr@,FilterProcPtr@)          TOOL 23,21
DEF FN    StopAlert%(AlertTemplatePtr@, FilterProcPtr@)     TOOL 24,21
DEF FN    NoteAlert%(AlertTemplatePtr@,FilterProcPtr@)      TOOL 25,21
DEF FN    CautionAlert%
              (AlertTemplatePtr@,
               FilterProcPtr@)                              TOOL 26,21
```

Manipulating Items

```
DEF PROC  ParamText
              (Param0String$,
               Param1String$,
               Param2String$,
               Param3String$)                               TOOL 27,21
DEF FN    GetControlDItem@(TheDialogPtr@,ItemID%)           TOOL 30,21
DEF PROC  GetIText(TheDialogPtr@,ItemID%,TextStringPtr@)TOOL 31,21
DEF PROC  SetIText(TheDialogPtr@,ItemID%,TextString$)       TOOL 32,21
DEF PROC  SelIText
              (TheDialogPtr@,
               ItemID%,
               StartSel%,
               EndSel%)                                     TOOL 33,21
DEF FN    GetDItemType%(TheDialogPtr@,ItemID%)              TOOL 38,21
DEF PROC  SetDItemType(ItemType%,TheDialogPtr@,ItemID%)     TOOL 39,21
DEF PROC  GetDItemBox
              (TheDialogPtr@,
               ItemID%,
               ItemBoxRectPtr@)                             TOOL 40,21
DEF PROC  SetDItemBox
              (TheDialogPtr@,
               ItemID%,
               ItemBoxRectPtr@)                             TOOL 41,21
DEF FN    GetFirstDItem%(TheDialogPtr@)                     TOOL 42,21
DEF FN    GetNextDItem%(TheDialogPtr@,ItemID%)              TOOL 43,21
DEF FN    GetDefButton%(TheDialogPtr@)                      TOOL 55,21
DEF PROC  SetDefButton(ItemID%,TheDialogPtr@)               TOOL 56,21
DEF FN    GetDItemValue%(TheDialogPtr@,ItemID%)             TOOL 46,21
DEF PROC  SetDItemValue
              (ItemValue%,
               TheDialogPtr@,
               ItemID%)                                     TOOL 47,21
DEF FN    GetAlertStage%                                    TOOL 52,21
DEF PROC  ResetAlertStage                                   TOOL 53,21
DEF FN      DefaultFilter%
                (TheDialogPtr@,
                 TheEventRecordPtr@,
                 ItemHitPtr%)                             TOOL 54,21
DEF PROC    HideDItem(TheDialogPtr@,ItemID%)              TOOL 34,21
DEF PROC    ShowDItem(TheDialogPtr@,ItemID%)              TOOL 35,21
DEF FN      FindDItem%(TheDialogPtr@, ThePointPtr@)       TOOL 36,21
DEF PROC    UpdateDialog(TheDialogPtr@,UpdateRgnHandle@)  TOOL 37,21
DEF PROC    DisableDItem(TheDialogPtr@,ItemID%)           TOOL 57,21
DEF PROC    EnableDItem(TheDialogPtr@,ItemID%)            TOOL 58,21
```

Event Manager

The Event Manager is responsible for handling the possible events which occur either from the user or from the computer. The Event Manager keeps track of all events and reports them to the application when requested to do so. Since some events have a higher priority than others, events are not reported in the same order that they occurred. A mouse down, a mouse click, double-click and disk insert are some examples of events.

Special Values

Value	Definition
Event Codes	
-1	Every Event
0	Null Event
1	Mouse Down
2	Mouse Up
3	Key Down
4	Undefined
5	Auto Key
6	Update Event
7	Undefined
8	Activate Event
9	Switch Event
10	Desk Accessory Event
11	Driver Event
12	Application Event #1
13	Application Event #2
14	Application Event #3
15	Application Event #4
Event Mask Equates	
2	Mouse Down Mask
4	Mouse Up Mask
8	Key Down Mask
32	Auto Key Mask
64	Update Mask
256	Activate Mask
1024	Switch Mask
2048	Driver Mask
4096	Application Mask #1
8192	Application Mask #2
16384	Application Mask #3
-32768	Application Mask #4

Modifier Flags

1	Active Flag
64	Button 1 State
128	Button 0 State
256	Apple Key
512	Shift Key
1024	Caps Lock
2048	Option Key
4096	Control Key
8192	Key Pad

Data Structures

EventRecord

The EventRecord is the datastructure used by the EventManager to report the occurrence of an event and appropriate related information.

```
DIM anEventRecord!(19)
```

Element(s)	Value	Description
0..1	Integer	Event code specifying which event occurred
2..5	Double Integer	Event message which has additional information about event
6..9	Double Integer	Number of ticks since startup
10..13	*Point*	Mouse location where event occurred
14..15	Integer	Modifier flags
16..17	Double Integer	Task Data for Task Master
18..19	Double Integer	Task Mask for Task Master

Routines

HouseKeeping

```
DEF PROC  EMStartUp
              (DPageAddr%,
               QueueSize%,
               XMinClamp%,
               XMaxClamp%,
               YMinClamp%,
               YMaxClamp%,
               UserID%)                          TOOL 3,6
DEF FN    EMVersion%                             TOOL 4,6
DEF PROC  EMReset                                TOOL 5,6
DEF FN    EMStatus%                              TOOL 6,6
```

Acessing Toolbox Events

```
DEF FN     GetNextEvent%(EventMask%,EventRecordPtr@)         TOOL 10,6
DEF FN     EventAvail%(EventMask%,EventRecordPtr@)           TOOL 11,6
```

Mouse Reading

```
DEF PROC   GetMouse(PointPtr@)                               TOOL 12,6
DEF FN     Button%(ButtonNum%)                               TOOL 13,6
DEF FN     StillDown%(ButtonNum%)                            TOOL 14,6
DEF FN     WaitMouseUp%(ButtonNum%)                          TOOL 15,6
```

Posting and Removing Events

```
DEF FN     PostEvent%(EventCode%,EventMsg@)                  TOOL 20,6
DEF FN     FlushEvents%(EventMask%,StopMask%)                TOOL 21,6
```

Accessing Operating System Events

```
DEF FN     GetOSEvent%(EventMask%,EventRecordPtr@)           TOOL 22,6
DEF FN     OSEventAvail%(EventMask%,EventRecordPtr@)         TOOL 23,6
```

Miscellaneous

```
DEF FN     TickCount@                                        TOOL 16,6
DEF FN     GetDblTime@,                                      TOOL 17,6
DEF FN     GetCaretTime@                                     TOOL 18,6
DEF PROC   SetEventMask(TheMask%)                            TOOL 24,6
DEF PROC   FakeMouse
               (Changed%,
                ModLatchByte%,
                XPosition%,
                YPosition%,
                ButtonStatus%)                               TOOL 25,6
DEF FN     DoWindows%                                        TOOL 9,6
DEF PROC   SetSwitch                                         TOOL 19,6
```

Font Manager

The Font Manager allows the application to use of different fonts, font styles, etc. within QuickDraw. Font definitions are found in the SYSTEM/FONTS/ directory of a bootable System Disk.

Special Values

No special values defined for the Font Manager.

Data Structures

Font

A font is a complete set of characters of one typeface or stylistic variation not including a size. Fonts are stored in the SYSTEM/FONTS/ directory of a System Disk. Manipulation of fonts by an application is usually accomplished through handles.

FontID

```
DIM aFontID!(3)
```

Element	Value	Definition
0..1	Integer	Font family number
2	Integer	Style
3	Integer	Point size of the font

FontStatRec

```
DIM aFontStatRec!(5)
```

Element	Value	Definition
0..3	*FontID*	Result ID
4..5	Integer	Result stats

Routines

HouseKeeping

```
DEF PROC  FMStartUp(UserID%,DPageAddr%)             TOOL 2, 27
DEF PROC  FMShutDown                                TOOL 3, 27
DEF FN    FMVersion%                                TOOL 4, 27
DEF PROC  FMReset                                   TOOL 5, 27
DEF FN    FMStatus%                                 TOOL 6, 27
```

Family Access

```
DEF FN    CountFamilies%(FamSpecs%)                 TOOL 9, 27
DEF FN    FindFamily%
             (FamSpecs%,
              PositionNum%,
              FamNameStringPtr@)                    TOOL 10, 27
DEF FN    GetFamInfo%(FamNum%,FamNameStringPtr@)    TOOL 11, 27
DEF FN    GetFamNum%(FamNameStringPtr@)             TOOL 12, 27
DEF PROC  AddFamily(FamNum%,FamNameString$)         TOOL 13, 27
```

Manipulation

```
DEF PROC  InstallFont(DesiredIDRecPtr@,ScaleWord%)  TOOL 14, 27
DEF PROC  SetPurgeStat(DesiredIDRecPtr@,PurgeStat%) TOOL 15, 27
DEF FN    CountFonts%(DesiredIDRecPtr@,Specs%)      TOOL 16, 27
DEF PROC  FindFontStats
             (DesiredIDRecPtr@,
              Specs%,
              PositionNum%,
              FontStatRecPtr@)                      TOOL 17, 27
DEF PROC  LoadFont
              (DesiredIDRecPtr@,
               Specs%,
               PositionNum%,
               FontStatRecPtr@)                     TOOL 18, 27
DEF PROC  LoadSysFont                               TOOL 19, 27
DEF PROC  AddFontVar(FontHndl@,NewSpecs%)           TOOL 20, 27
```

Menu and Dialog Fonts

```
DEF PROC  FixFontMenu(MenuID%,StartingID%,FamSpecs%)  TOOL 21, 27
DEF FN    ChooseFont@(currentIDRecPtr@,FamSpecs%)     TOOL 22, 27
DEF FN    ItemID2FamNum%(itemID%)                     TOOL 23, 27
```

Miscellaneous

```
DEF PROC  FMSetSysFont(TheFontIDRecPtr@)            TOOL 24, 27
DEF FN    FMGetSysFID@                              TOOL 25, 27
DEF FN    FMGetCurFID@                              TOOL 26, 27
```

Integer Math

The Integer Math tool set consists of a varied collection of operations for integers, long integers and signed fractional numbers. These operations include multiplication, division and various conversions between numeric representations and strings.

Special Values

No special values defined for Integer Math.

Data Structures

No data structures defined for Integer Math.

Routines

Housekeeping Routines

```
DEF PROC IMStartUp                                                TOOL 2,11
DEF PROC IMShutDown                                               TOOL 3,11
DEF FN   IMVersion%                                               TOOL 4,11
DEF PROC IMReset                                                  TOOL 5,11
DEF FN   IMStatus%                                                TOOL 6,11
```

Math Routines

```
DEF FN   Muliply@(Left%,Right%)                                   TOOL 9,11
DEF FN   SDivide@(Left%,Right%)                                   TOOL 10,11
DEF FN   UDivide@(Left%,Right%)                                   TOOL 11,11
DEF FN   LongDivide@[2](Left@,Right@)                             TOOL 13,11
DEF FN   FixRatio@(Numerator%,Denominator%)                       TOOL 14,11
DEF FN   FixMul@(Left@,Right@)                                    TOOL 15,11
DEF FN   FracMul@(Left@,Right@)                                   TOOL 16,11
DEF FN   FixDiv@(Dividend@,Divisor@)                              TOOL 17,11
DEF FN   FracDiv@(Dividend@,Divsor@)                              TOOL 18,11
DEF FN   FixRound%(Fixed@)                                        TOOL 19,11
DEF FN   FracSqrt@(Frac@)                                         TOOL 20,11
DEF FN   FracCos@(Fixed@)                                         TOOL 21,11
DEF FN   FracSin@(Fixed@)                                         TOOL 22,11
DEF FN   FixATan2@(Val1@,Val2)                                    TOOL 23,11
DEF FN   HiWord%(Long@)                                           TOOL 24,11
DEF FN   LoWord%(Long@)                                           TOOL 25,11
```

Numeric Conversion Routines

```
DEF FN    Long2Fix@(Long@)                                 TOOL 26,11
DEF FN    Fix2Long@(Fixed@)                                TOOL 27,11
DEF FN    Fix2Frac@(Fixed@)                                TOOL 28,11
DEF FN    Frac2Fix@(Frac@)                                 TOOL 29,11
DEF PROC  Fix2X(Fixed@,ExtPtr@)                            TOOL 30,11
DEF PROC  Frac2X(Frac@,ExtPtr@)                            TOOL 31,11
DEF FN    X2Fix@(ExtPtr@)                                  TOOL 32,11
DEF FN    X2Frac@(ExtPtr@)                                 TOOL 33,11
```

String Conversion Routines

```
DEF PROC  Int2Hex(Int%,BufferPtr@,BufferLen%)              TOOL 34,11
DEF PROC  Long2Hex(Long%,BufferPtr@,BufferLen%)            TOOL 35,11
DEF FN    Hex2Int%(BufferPtr@,BufferLen%)                  TOOL 36,11
DEF FN    Hex2Long@(BufferPtr@,BufferLen%)                 TOOL 37,11
DEF PROC  Int2Dec(Int%,BufferPtr@,BufferLen%,Signed%)      TOOL 38,11
DEF PROC  Long2Dec(Long%,BufferPtr@,BufferLen%,Signed%)    TOOL 39,11
DEF FN    Dec2Int%(BufferPtr@,BufferLen%,Signed%)          TOOL 40,11
DEF FN    Dec2Long@(BufferPtr@,BufferLen%,Signed%)         TOOL 41,11
DEF FN    HexIt@(Int%)                                     TOOL 42,11
```

Line Edit

Line Edit is used to display a line of text on the screen and allow a user to edit the text. The editing operations include the standard cut, copy and paste operations.

Special Values

No special values defined for Line Edit.

Data Structures

LERec

The Line Edit Record data structure holds the information needed to store and manipulate a line of text in memory. A program should not access the LERec data structure directly, but only through the use of the Line Edit routines.

```
DIM aLERec!(52)
```

Element	Value	Definition
0..3	Double Integer	Handle to the text to be edited
4..6	Integer	Length of the text
7..14	*Rect*	Destination rectangle
15..22	*Rect*	View rectangle
23..26	Double Integer	Pointer to GrafPort
27..28	Integer	Used for highlighting
29..30	Integer	Used for drawing the text
31..32	Integer	Start of selection range
33..34	Integer	End of selection range
35..36	Integer	Used internally
37..38	Integer	Used internally
39..40	Integer	Used internally
41..44	Double Integer	Used internally
45..48	Double Integer	Pointer to highlight routine: *Proc*Ptr
49..52	Double Integer	Pointer to caret routine: *Proc*Ptr

Routines

HouseKeeping

```
DEF PROC  LEStartUp(DPageAddr%,UserID%)               TOOL 2,20
DEF PROC  LEShutDown                                  TOOL 3,20
DEF FN    LEVersion%                                  TOOL 4,20
DEF PROC  LEReset                                     TOOL 5,20
DEF FN    LEStatus%                                   TOOL 6,20
```

Creating and Disposing

```
DEF FN     LENew@(DestRectPtr@,ViewRectPtr@,MaxTextLen%) TOOL 9,20
DEF PROC   LEDispose(LERecHndl@)                         TOOL 10,20
```

Changing the Text of an Edit Record

```
DEF PROC   LESetText(TextPtr@, Length%,LERecHndl@)       TOOL 11,20
```

Insertion Point and Selection Range

```
DEF PROC   LEIdle(LERecHndl@)                            TOOL 12,20
DEF PROC   LEClick(EventRecordPtr@,LERecHndl@)           TOOL 13,20
DEF PROC   LESetSelect(SelStart%,SelEnd%,LERecHndl@)     TOOL 14,20
DEF PROC   LEActivate(LERecHndl@)                        TOOL 15,20
DEF PROC   LEDeactivate(LERecHndl@)                      TOOL 16,20
```

Editing

```
DEF PROC   LEKey(TheKey%,Modifiers%,LERecHndl@)          TOOL 17,20
DEF PROC   LECut(LERecHndl@)                             TOOL 18,20
DEF PROC   LECopy(LERecHndl@)                            TOOL 19,20
DEF PROC   LEPaste(LERecHndl@)                           TOOL 20,20
DEF PROC   LEDelete(LERecHndl@)                          TOOL 21,20
DEF PROC   LEInsert(TextPtr@,Length%,LERecHndl@)         TOOL 22,20
```

Text Display

```
DEF PROC   LEUpdate(LERecHndl@)                          TOOL 23,20
DEF PROC   LETextBox
               (TextPtr@,
                Length%,
                BoxRectPtr@,
                TextJustify%)                            TOOL 24,20
```

Scrap Handling

```
DEF PROC   LEFromScrap                                   TOOL 25,20
DEF PROC   LEToScrap                                     TOOL 26,20
DEF FN     LEScrapHandle@                                TOOL 27,20
DEF FN     LEGetScrapLen%                                TOOL 28,20
DEF PROC   LESetScrapLen(NewLength%)                     TOOL 29,20
```

Setting HiliteHook and CaretHook

```
DEF PROC   LESetHilite(HiliteProcPtr@,LERecHndl@)        TOOL 30,20
DEF PROC   LESetCaret(CaretProcPtr@,LERecHndl@)          TOOL 31,20
```

List Manager

The List Manager is used to create, display and allow selection of a variable amount of similar data.

Special Values

No special values defined for Line Edit.

Data Structures

ListMem

The ListMem structure holds the data for one member in a list. Custom members can be defined by adding more data to the end of the existing data. The list contains a field that holds the number of bytes in one member.

```
DIM aListMem!(N)
```

Element	Value	Definition
0..3	Double Integer	Pointer to string for string member, anything else for custom
4..5	Integer	Bit Flags, bit 7=1 if selected, bit 6 = 1 if disabled
•	•	•
•	•	•
•	•	•
N	Integer	Application data if custom member

ListRec

The ListRec data structure is where information is kept about a List Manager list. Although the List Manager provides routines to manipulate the list, the actual structure is provided below.

```
DIM aListRec!(35)
```

Element	Value	Definition
0..7	*Rect*	Bounding rectangle
8..9	Integer	Total number of members in the list
10..11	Integer	Number of members displayed at any one time
12..13	Integer	List type
14..15	Integer	The number of the member appearing at start up
16..19	Double Integer	Handle of control belonging to the list
20..23	Double Integer	Pointer to list's draw procedure : *Proc*Ptr
24..25	Integer	Height of each member in pixels
26..27	Integer	Number of bytes in a member record
28..31	Double Integer	Pointer to member list which is array of records
32..35	Double Integer	Pointer to scroll bar color table

Routines

HouseKeeping

```
DEF PROC  ListStartUp                                       TOOL 2,28
DEF PROC  ListShutDown                                      TOOL 3,28
DEF FN    ListVersion%                                      TOOL 4,28
DEF PROC  ListReset                                         TOOL 5,28
DEF FN    ListStatus%                                       TOOL 6,28
```

Manipulation

```
DEF FN    CreateList@(TheWindowPtr@, ListRecPtr@)           TOOL 9,28
DEF FN    NextMember@(ListMemPtr@, ListRecPtr@)             TOOL 11,28
DEF FN    ResetMember@(ListRecPtr@)                         TOOL 15,28
DEF PROC  DrawMember(ListMemPtr@,ListRecPtr@)               TOOL 12,28
DEF PROC  SelectMember(ListMemPtr@,ListRecPtr@)             TOOL 13,28
DEF PROC  SortList(SortProcPtr@, ListRecPtr@)               TOOL 10,28
DEF FN    GetListDefProc@                                   TOOL 14,28
DEF PROC  NewList(ListMemPtr@,ListRecPtr@)                  TOOL 16,28
```

Memory Manager

The Memory Manager is one of the most important tool sets in the Toolbox. This tool is entirely responsible for the allocation, deallocating, and repositioning of memory blocks on the Apple IIGS. The Memory Manager keeps track of how much memory is free and what parts are allocated and to whom. Whenever a program needs memory, it must ask the Memory Manager to allocate it.

Special Values

The following values are used with the NewHandle@ function to specify the attributes of a block of memory to be allocated.

Value	Attribute	Definition
1	Fixed Bank	must be in a particular bank of memory
2	Fixed Address	must be allocated at specific address
4	Page Aligned	must be page aligned
8	Special Memory	may not use bank \$00, \$01, \$E0, \$E1
16	No Bank Cross	may not cross bank boundary
16384	Fixed Block	can never be move
32768	Locked	same as HLock

Data Structures

Pointer (Ptr)

A pointer in TML BASIC is the address of a array, structure or simple variable. The address of a variable is obtained using the VARPTR function. A pointer is stored as a double integer value.

Handle (Hndl)

A handle is also a pointer in TML BASIC, but behaves in a very special way. A handle always points to another pointer which in turn points to a variable. Except for very rare cases should attempt to create a handle to be used with the Toolbox using the VARPTR function. A program should only obtain handle values by calling the NewHandle@ function or some other Toolbox routine which returns a handle value.

Routines

HouseKeeping

```
DEF FN      MMStartUp%                                TOOL 2,2
DEF PROC    MMShutDown%(UserID%)                      TOOL 3,2
DEF FN      MMVersion%                                TOOL 4,2
DEF PROC    MMReset                                   TOOL 5,2
DEF FN      MMStatus%                                 TOOL 6,2
```

Memory Allocation

```
DEF FN      NewHandle@
                (BlockSize@,
                 UserID%,
                 Attributes%,
                 LocationPtr@)                        TOOL 9,2
DEF PROC    ReAllocHandle
                 (oldHandle@,
                  BlockSize@,
                  UserID%,
                  Attributes%,
                  LocationPtr@)                       TOOL 10,2
DEF PROC    RestoreHandle(Handle@)                    TOOL 11,2
DEF PROC    DisposeHandle(Handle@)                    TOOL 16,2
DEF PROC    DisposeAll(UserID%)                       TOOL 17,2
DEF PROC    PurgeHandle(Handle)                       TOOL 18,2
DEF PROC    PurgeAll(UserID%)                         TOOL 19,2
```

Size Information

```
DEF FN      GetHandleSize@(Handle@)                   TOOL 24,2
DEF PROC    SetHandleSize(NewSize@,Handle@)           TOOL 25,2
DEF FN      FindHandle@(LocationPtr@)                 TOOL 26,2
```

Locking and Purge Level

```
DEF PROC    HLock(Handle@)                            TOOL 32,2
DEF PROC    HLockAll(UserID%)                         TOOL 33,2
DEF PROC    HUnLock(Handle@)                          TOOL 34,2
DEF PROC    HUnLockAll(UserID%)                       TOOL 35,2
DEF PROC    SetPurge(NewPurgeLevel%,Handle)           TOOL 36,2
DEF PROC    SetPurgeAll(NewPurgeLevel%,UserID%)       TOOL 37,2
```

Free Space

```
DEF PROC    CompactMem                                TOOL 31,2
DEF FN      FreeMem@                                  TOOL 27,2
DEF FN      MaxBlock@                                 TOOL 28,2
DEF FN      TotalMem@                                 TOOL 29,2
DEF PROC    CheckHandle(Handle@)                      TOOL 30,2
```

Miscellaneous

```
DEF PROC  BlockMove(SrcPtr@,DstPtr@,ByteCount@)          TOOL 43,2
DEF PROC  PtrToHand(SrcPtr@,DstHandle@,ByteCount@)       TOOL 40,2
DEF PROC  HandToPtr(SrcHandle@,DstPtr@,ByteCount@)       TOOL 41,2
DEF PROC  HandToHand(SrcHandle@,DstHandle@,ByteCount@)   TOOL 42,2
```

Menu Manager

The Menu Manager is responsible for the creation, manipulation and disposal of the pull down menus used in the desktop environment.

Special Values

No special values defined for the Menu Manager.

Data Structures

MenuBar

The MenuBar data structure contains the information needed to manage the standard desktop menu bar as well as the menus contained within the menu bar. This is a private data structure and should only be manipulated by calling the Menu Manager routines.

Menu

A Menu data structure contains the information needed to manage a single pull-down menu. Again, this data structure is private and should only be manipulated by calling the Menu Manager routines.

Routines

HouseKeeping

```
DEF PROC  MenuStartUp(UserID%,DPageAddr%)            TOOL 2,15
DEF PROC  MenuShutDown                               TOOL 3,15
DEF FN    MenuVersion%                               TOOL 4,15
DEF PROC  MenuReset                                  TOOL 5,15
DEF FN    MenuStatus%                                TOOL 6,15
```

Creating and Disposing

```
DEF FN    NewMenuBar@(TheWindowPtr@)                 TOOL 21,15
DEF FN    NewMenu@(MenuStringPtr@)                   TOOL 45,15
DEF PROC  DisposeMenu(TheMenuHndl@)                  TOOL 46,15
DEF FN    FixMenuBar%,                               TOOL 19,15
DEF PROC  CalcMenuSize(NewWidth%,NewHeight%,MenuNum%) TOOL 28,15
```

User Interaction

```
DEF PROC  MenuSelect(EventRecordPtr@,TheMenuBarHndl@) TOOL 43,15
DEF PROC  MenuKey(EventRecordPtr@,TheMenuBarHndl@)   TOOL 9,15
DEF PROC  MenuRefresh(RedrawRoutineProcPtr@)         TOOL 11,15
DEF PROC  DrawMenuBar                                TOOL 42,15
DEF PROC  HiliteMenu(Hilite%,MenuNum%)               TOOL 44,15
DEF PROC  FlashMenuBar                                    TOOL 12,15
```

Menu and Item Shuffling

```
DEF PROC  InsertMenu(AddMenuHandle@,InsertAfter%)         TOOL 13,15
DEF PROC  DeleteMenu(MenuNum%)                            TOOL 14,15
DEF PROC  InsertMItem
              (AddItemCStringPtr@,
               InsertAfter%,
               MenuNum%)                                  TOOL 15,15
DEF PROC  DeleteMItem(ItemNum%)                           TOOL 16,15
```

Menu Bar Access

```
DEF PROC  SetSysBar(TheMenuBarHndl@)                      TOOL 18,15
DEF FN    GetSysBar@                                      TOOL 17,15
DEF PROC  SetMenuBar(TheMenuBarHndl@)                     TOOL 57,15
DEF FN    GetMenuBar@                                     TOOL 10,15
DEF PROC  SetBarColors
              (NewBarColor%,
               NewInvertColor%,
               NewOutlineColor%)                          TOOL 23,15
DEF FN    GetBarColors@                                   TOOL 24,15
DEF PROC  SetMTitleStart(XStart%)                         TOOL 25,15
DEF FN    GetMTitleStart%                                 TOOL 26,15
DEF FN    CountMItems%(MenuNum%)                          TOOL 20,15
```

Menu Record Access

```
DEF FN    GetMHandle@(MenuNum%)                           TOOL 22,15
DEF PROC  SetMTitleWidth(NewWidth%,MenuNum%)              TOOL 29,15
DEF FN    GetMTitleWidth%(MenuNum%)                       TOOL 30,15
DEF PROC  SetMenuFlag(NewValue%,MenuNum%)                 TOOL 31,15
DEF FN    GetMenuFlag%(MenuNum%)                          TOOL 32,15
DEF PROC  SetMenuTitle(NewTitleStringPtr@,MenuNum%)       TOOL 33,15
DEF FN    GetMenuTitle@(MenuNum%)                         TOOL 34,15
DEF PROC  SetMenuID(NewMenuNum%,OldMenuNum%)              TOOL 55,15
```

Item Record Access

```
DEF PROC  SetMItem(NewStrgCStringPtr@,ItemNum%)           TOOI 36,15
DEF FN    GetMItem@(ItemNum%)                             TOOL 37,15
DEF PROC  SetMItemName(ItemTitleStringPtr@,ItemNum%)      TOOL 58,15
DEF PROC  EnableMItem(ItemNum%)                           TOOL 48,15
DEF PROC  DisableMItem(ItemNum%)                          TOOL 49,15
DEF PROC  CheckMItem(Checked%,ItemNum%)                   TOOL 50,15
DEF PROC  SetMItemMark(MarkChar%,ItemNum%)                TOOL 51,15
DEF FN    GetMItemMark%(ItemNum%)                         TOOL 52,15
DEF PROC  SetMItemStyle (ChStyle%,ItemNum%)               TOOL 53,15
DEF FN    GetMItemStyle%(ItemNum%)                        TOOL 54,15
DEF PROC  SetMItemFlag(NewValue%,ItemNum%)                TOOL 38,15
DEF FN    GetMItemFlag%(ItemNum%)                         TOOL 39,15
DEF PROC  SetMItemID(NewID%,ItemNum%)                     TOOL 56,15
DEF PROC   SetMItemBlink(Count%)                       TOOL 40,15
```

Miscellaneous

```
DEF FN     GetMenuMgrPort@                             TOOL 27,15
DEF PROC   MenuNewRes                                  TOOL 41,15
DEF PROC   InitPalette                                 TOOL 47,15
DEF FN     MenuGlobal%(chgFlag%)                       TOOL 35,15
```

Miscellaneous Tools

The Miscellaneous Tool Set consists mostly of system level routines that must be available for most other tool sets. The tool set includes operations to read and write the Apple IIGS realtime clock, read and write the parameter memory, low level mouse operations, interrupts and others.

Special Values

No special values defined for the Miscellaneous Tool Set.

Data Structures

No special data structures defined for the Miscellaneous Tool Set.

Routines

HouseKeeping

```
DEF PROC  MTStartUp                                           TOOL 2,3
DEF PROC  MTShutDown                                          TOOL 3,3
DEF FN    MTVersion%                                          TOOL 4,3
DEF PROC  MTReset                                             TOOL 5,3
DEF FN    MTStatus%                                           TOOL 6,3
```

Battery RAM

```
DEF PROC  WriteBRam(BufferPtr@)                               TOOL 9,3
DEF PROC  ReadBRam(BufferPtr@)                                TOOL 10,3
DEF PROC  WriteBParam(Data%,ParamRefNum%)                     TOOL 11,3
DEF FN    ReadBParam%(ParamRefNum%)                           TOOL 12,3
```

Clock

```
DEF FN    ReadTimeHex%[4](BufferSize%)                        TOOL 13,3
DEF PROC  WriteTimeHex(MonthDay%,YearHour%,MinuteSecond%)     TOOL 14,3
DEF PROC  ReadAsciiTime(BufferPtr@)                           TOOL 15,3
```

Vector Initialization

```
DEF PROC  SetVector(VectorRefNum%,VectorProcPtr@)             TOOL 16,3
DEF FN    GetVector@(VectorRefNum%)                           TOOL 17,3
```

HeartBeat

```
DEF PROC  SetHeartBeat(TaskProcPtr@)                          TOOL 18,3
DEF PROC  DelHeartBeat(TaskProcPtr@)                          TOOL 19,3
DEF PROC  ClrHeartBeat                                        TOOL 20,3
```

System Death Manager

```
DEF PROC   SysFailMgr(ErrCode%,MsgString$)                 TOOL 21,3
```

Get Address

```
DEF FN     GetAddr@(RefNum%)                               TOOL 22,3
```

Mouse

```
DEF FN     ReadMouse!(6)                                   TOOL 23,3
DEF PROC   InitMouse(MouseSlot%)                           TOOL 24,3
DEF PROC   SetMouse(MouseMode%)                            TOOL 25,3
DEF PROC   HomeMouse                                       TOOL 26,3
DEF PROC   ClearMouse                                      TOOL 27,3
DEF PROC   ClampMouse
               (XMinClamp%,
                XMaxClamp%,
                YMinClamp%,
                YMaxClamp%)                                TOOL 28,3
DEF FN     GetMouseClamp%[4]                               TOOL 29,3
DEF PROC   PosMouse(XPos%,YPos%)                           TOOL 30,3
DEF FN     ServeMouse%                                     TOOL 31,3
DEF FN     GetNewID%(IDTag%)                               TOOL 32,3
DEF PROC   DeleteID(IDTag%)                                TOOL 33,3
DEF PROC   StatusID(IDTag%)                                TOOL 34,3
```

Interrupt Control

```
DEF PROC   IntSource(ScrRefNum%)                           TOOL 35,3
```

Tick Count

```
DEF FN     GetTick@                                        TOOL 37,3
```

PackBytes and UnPackBytes

```
DEF FN     PackBytes%
               (ScrBufferPtr@,
                ScrSizePtr@,
                DstBufferPtr@,
                DstSize%)                                  TOOL 38,3
DEF FN     UnPackBytes%
               (ScrBufferPtr@,
                ScrSize%,
                DstBufferPtr@,
                DstSizePtr@)                               TOOL 39,3
```

Munger

```
DEF PROC  Munger
              (DestPtr@,
               DestLenPtr@,
               TargPtr@,
               TargLen%,
               ReplacePtr@,
               ReplaceLen%,
               PadCharPtr@)                              TOOL 40,3
```

Interrupt Enable State

```
DEF FN    GetIRQEnable%                                  TOOL 41,3
DEF PROC  SetAbsClamp
              (XMinClamp%,
               XMaxClamp%,
               YMinClamp%,
               YMaxClamp%)                               TOOL 42,3
DEF FN    GetAbsClamp%[4]                                TOOL 43,3
DEF PROC  SysBeep                                        TOOL 44,3
```

Note Synthesizer

The Note Synthesizer is a sophisticated tool set which can generate complex musical sounds based upon instrument definitions. The Ensoniq Digital Oscillator Chip (DOC) is the hardware that actually creates the sound.

Special Values

No special values defined for the Note Synthesizer.

Data Structures

WaveForm

The WaveForm data structure defines a wave which contains information about the allowable pitch and range of a wave. A wave is a component of a wave list which is needed for defining an Instrument data structure. A wave list consists of a list of waves.

```
DIM aWaveForm!(5)
```

Element	Value	Definition
0	Integer	The highest MIDI semi-tone, also named TopKey.
1	Integer	Wave Address to be put in DOC registers
2	Integer	Wave Size
3	Integer	DOC Mode
4..5	Integer	Relative Pitch for tuning the waveform

Instrument

The Instrument data structure is used with the NoteOn procedure to define the characteristics of an instrument whose sounds are created by the Note Synthesizer. This structure has a variable length which depends on the number of waveforms in both Wave List A and Wave List B. The LA and LB notation refers to the "Last element in wave A" and "Last element in wave B" respectively.

```
DIM anInstrument!(LB)
```

Element(s)	Value	Definition
0..23	Envelope!(23)	Envelope
24	Integer	Release Segment
25	Integer	Priority Increment
26	Integer	Pitch bend range
27	Integer	Vibrato Depth
28	Integer	Vibrato Speed
29	Integer	Spare, not used
30	Integer	Number of WaveForm structures in WaveList A
31	Integer	Number of WaveForm structures in WaveList B
32..LA	WaveA!(LA)	WaveList A
LA+1..LB	WaveB!(LB-LA)	WaveList B

Routines

HouseKeeping

```
DEF PROC  NSStartUp(UpdateRate%,UpdateProcPtr@)               Tool 2,25
DEF PROC  NSShutDown                                          Tool 3,25
DEF FN    NSVersion%                                          Tool 4,25
DEF PROC  NSReset                                             Tool 5,25
DEF FN    NSStatus%                                           Tool 6,25
```

Generator Allocation

```
DEF FN    AllocGen%(RequestPriority%)                         Tool 9,25
DEF PROC  DeallocGen(GenNumber%)                              Tool 10,25
```

Note Manipulation

```
DEF PROC  NoteOn(GenNum%,SemiTone%,Volume%,InstrumentPtr@)    Tool 11,25
DEF PROC  NoteOff(GenNum%,SemiTone%)                          Tool 12,25
DEF PROC  AllNotesOff                                         Tool 13,25
```

Print Manager

The Print Manager transforms Quickdraw representations of documents into printed form. A special print driver is required to use the Print Manager. At the moment Print Manager drivers exist only for the ImageWriter and the LaserWriter printers.

Special Values

Value	Definition
0	Specify draft printing
128	Specify spool printing
16382	Maximum number of pages in a spool file
0	The NoError error code

Data Structures

TPrPort

TPrPort specifies the port that the Print Manager uses as it's printing environment. This data structure is nearly the same as Quickdraw's GrafPort data structure. In fact, the first element in the TPrPort structure is a GrafPort. The extra fields in the TPrPort are private and should only be changed using the Print Manager routines.

TPrInfo

The TPrInfo data structure is the printer information record which holds page composition information.

```
DIM  aTPrInfo%(6)
```

Element	Value	Definition
0	Integer	Used internally
1	Integer	Vertical resolution of printer
2	Integer	Horizontal resolution of printer
3..6	*Rect*	Page definition rectangle

TPrStl

The TPrStl data structure defines the style information obtained from the user via the style dialog as well as the job dialog. The fields in this record have different meanings for different printers.

```
DIM  aTPrStl%(6)
```

Element	Value	Definition
0	Integer	Used internally
1	Integer	Vertical resolution of printer
2	Integer	Horizontal resolution of printer
3..6	*Rect*	Page definition rectangle

TPrXInfo

The TPrXInfo data structure contains extra information that an application may require.

```
DIM  aTPrXInfo%(N)
```

Element	Value	Definition
0	Integer	Used internally
1	Integer	Reserved for internal use
2	Integer	Reserved for internal use
3	Integer	Size in byte of buffer used for spool printing
4	Integer	Additional information for internal use
•	•	•
•	•	•
•	•	•
N	Integer	End of additional information for internal use

TPrJob

The TPrJob contains pertinent information about one particular printing job. Its contents are set as a result of the job dialog.

```
DIM  aTPrJob!(11)
```

Element	Value	Definition
0..1	Integer	First page to print
2..3	Integer	Last page to print
4..5	Integer	Number of copies to print
6	Integer	Printing method (ex: 0 = draft printing)
7	Integer	Used internally
8..11	Double Integer	Pointer to a background process: *ProcPtr*

TPrint

The TPrint data structure is used primarily for grouping most of the other Print Manager data structures in one place. This data structure is of variable size since theTPrXInfo is variable size.

```
DIM  aTPrint!(I+29)
```

Element	Value	Definition
0..1	Integer	Print manager version
2.. 15	*TPrInfo*	Printer information subrecord
16..23	*Rect*	Paper rectangle
24..37	*TPrStl*	Print style information
38..51	*TPrInfo*	Used internally
52..I	*TPrXInfo*	Additional information
I+1..I+11	*TPrJob*	Print job subrecord
I+12..I+29	Buffer!(18)	Generic data structure not used

```
DIM  aTPrStatus!(22)
```

Element	Value	Definition
0..1	Integer	Total pages in spool file
2..3	Integer	Current page being printed
3..4	Integer	Total number of copies requested
5..6	Integer	Used internally
7..8	Integer	Used internally
9	Integer	Boolean value that's true if page has started printing
10	Integer	Used internally
11..14	Double Integer	Handle to TPrint print record
15..18	Double Integer	Pointer to TPrPort
19..22	Double Integer	Handle to a picture: *Pic*Hndl

Routines

Housekeeping

```
DEF PROC   PMStartUp(UserID%,DPageAddr%)                          TOOL 2,19
DEF PROC   PMShutDown                                             TOOL 3,19
DEF FN     PMVersion%                                             TOOL 4,19
DEF PROC   PMReset                                                TOOL 5,19
DEF FN     PMStatus%                                              TOOL 6,19
```

Print Records and Dialogs

```
DEF PROC   PrDefault(THPrintHndl@)                                TOOL 9,19
DEF FN     PrValidate%(THPrintHndl@)                              TOOL 10,19
DEF FN     PrStlDialog%(THPrintHndl@)                             TOOL 11,19
DEF FN     PrJobDialog%(THPrintHndl@)                             TOOL 12,19
DEF FN     PrChoosePrinter%                                       TOOL 22,19
```

Printing

```
DEF FN     PrOpenDoc@(THPrintHandle@,TPPrPort@)                   TOOL 14,19
DEF PROC   PrCloseDoc(TPPrPort@)                                  TOOL 15,19
DEF PROC   PrOpenPage(TPPrPort@,PageFrameTPRectPtr@)              TOOL 16,19
DEF PROC   PrClosePage(TPPrPort@)                                 TOOL 17,19
DEF PROC   PrPicFile(THPrintHndl@,TPPrPort@,TPPrStatus@)          TOOL 18,19
```

Error Handling

```
DEF FN     PrError%                                               TOOL 20,19
DEF PROC   PrSetError(iErr%)                                      TOOL 21,19
```

QuickDraw

QuickDraw is the tool set that controls the graphics environment of the Apple IIGS and draws simple objects and text in the Super Hi-Res grahpics screen. All other tools which create graphical objects such as the Menu and Window Manager call the QuickDraw tool set.

Special Values

Transfer Modes

Transfer modes determine how bits are finally displayed when placing an image over or on top of another image.

Value	Description of Value
0	srcCopy
1	srcOr
2	srcXor
3	srcBic
32768	notSrcCopy
32769	notSrcOr
32770	notSrcXor
32771	notSrcBic

Special Text Transfer Modes

The following modes are exclusively used for text transfer.

Value	Description of Value
4	foreCopy
5	foreOr
6	foreXor
7	foreBic
32772	notforeCopy
32773	notforeOr
32774	notforeXor
32775	notforeBic

Text Styles

The type style of text characters is determined by the following special values. Some fonts, including the system font, may not support all the given styles or attributes.

Value	Description of Value
0	plain
1	bold
2	italic
4	underline
8	outline
16	shadow

Data Structures

BufSizeRec

The BufSizeRec data structure is used to store information that Quickdraw uses to manipulate its internal text buffers.

```
DIM BufSizeRec%(3)
```

Element(s)	Value	Description
0	Integer	Maximum width
1	Integer	Text buffer height
2	Integer	Text buffer row of words
3	Integer	Font width

ColorTable

The ColorTable data structure is used to hold information that specifies color intensities. The table is composed of 16 2-byte entries. The 2 bytes of an entry are divided into four 4 bit nybbles, of which only the low 3 are used. The lowest nybble holds the intensity of the color blue, the next nybble specifies the intensity of the color green and the third nybble indicates the intensity of the color red. Apple has reserved the high nybble of the high byte for future use. The actual colors achieved depends on the resolution mode (320 or 640) and also upon the dithering techniques used.

```
DIM aColorTable!(31)
```

Element(s)	Value	Description
0..1	Integer	Entry #1
2..3	Integer	Entry #2
4..5	Integer	Entry #3
•	•	•
•	•	•
•	•	•
30..31	Integer	Entry #16

CString

The CString data structure refers to the way the language C specifies a string in memory. A CString is different from a Pascal counted string in that the CString does not have a length byte and so does not indicate the number of characters in the string. Instead of the length byte, the CString has a termination character to specify where the last character in the string is stored. The termination character is a zero. Whenever QuickDraw or another toolbox routine refers to this string convention, it will explicitly state in the documentation, "CString".

```
DIM aCString!(N)
```

Element(s)	Value	Description
0	Integer	ASCII value of character #1 in string
1	Integer	ASCII value of character #2 in string
•	•	•
•	•	•
•	•	•
N-1	Integer	ASCII value of LAST character in string
N	Integer	termination character must be zero (0).

Cursor

The Cursor data structure is a variable length structure that defines a QuickDraw cursor which is used to indicate the current position of the mouse. The cursor definition bytes and the cursor mask should be specified row by row. The *Hot Spot* coordinates indicate the position in the rectangle that is aligned with the mouse position; for example, the hot spot of the arrow cursor is the tip of the arrow.

```
DIM aCursor!(P+3)
```

Element(s)	Value	Description
0..1	Integer	Height (in rows) of rectangle enclosing cursor
2..3	Integer	Width (in words) of rectangle enclosing cursor
4	Integer	Byte #1 of cursor definition
5	Integer	Byte #2 of cursor definition
•	•	•
•	•	•
•	•	•
R	Integer	Byte #R of cursor definition
R+1	Integer	Byte #1 of cursor mask
R+2	Integer	Byte #2 of cursor mask
•	•	•
•	•	•
•	•	•
P..P+1	Integer	Vertical position of mouse *Hot Spot*
P+2..P+3	Integer	Horizontal position of mouse *Hot Spot*

FontInfoRec

The FontInfoRec data structure is used to hold information regarding the current font.

```
DIM aFontInfoRec%(3)
```

Element(s)	Value	Description
0	Integer	Ascent : the number of pixel rows above the baseline
1	Integer	Descent : the number of pixel rows below the baseline
2	Integer	Maximum character width of any character
3	Integer	Leading : recommended number of rows between ascent and descent

FontGlobalsRec

The FontGlobalsRec data structure is a variable length structure that holds information about the current font. The data structure has a dynamic size to allow for future expansion. The size is returned by the the QuickDraw routine GetFGSize. The current elements of this data structure are defined as integers although the implementation could change in the future.

```
DIM aFontGlobalsRec%(N)
```

Element(s)	Value	Description
0	Integer	Family number
1	Integer	Style
2	Integer	Size
3	Integer	Version
4	Integer	Maximum width
5	Integer	Font bounds rectangle extent
•	•	•
•	•	•
•	•	•
N	Integer	Additional fields that Apple may add.

GrafPort

The current definition of the drawing environment is stored in the GrafPort data structure. Although the structure contents are defined here, the use of the Quickdraw routines to manipulate the GrafPort is strongly recommended. A GrafPort is analogous to the artist's palette or the draftman's drawing board. There can be more than one GrafPort on the desktop at one time, each with its own environment settings.

```
DIM aGrafPort!(169)
```

Element(s)	Value	Description
0..15	*LocInfoRec*	Quickdraw drawing location characteristics
16..23	*Rect*	Port rectangle
24..27	Double Integer	Clip region handle
28..31	Double Integer	Visible region handle
32..63	*Pattern*	Background pattern
64..67	*qui*	Pen location
68..71	*Point*	Pen size
72..73	Integer	Pen mode
74..105	*Pattern*	Pen pattern
106..113	*Mask*	Pen mask
114..115	Integer	Pen visible code
116..119	Double Integer	Font handle
120..123	Double Integer	Font ID
124..125	Integer	Font flags
126..127	Integer	Text size
128..129	Integer	Text face
130..131	Integer	Text mode
132..135	Double Integer	Space extra

136..139	Double Integer	Character extra
140..141	Integer	Foreground color
142..143	Integer	Background color
144..147	Double integer	Picture save handle
148..151	Double Integer	Region save handle
152..155	Double Integer	Polygon save handle
156..159	Double Integer	QDProcs pointer
160..161	Integer	Arc rotation
162..165	Double Integer	User field
166..169	Double Integer	System field

LocInfoRec

The LocInfoRec data structure holds characteristic information about a specific area of memory that Quickdraw can use as its drawing area.

```
DIM aLocInfoRec!(15)
```

Element(s)	Value	Description
0	Integer	Scanline control byte (SCB)
1	Integer	Reserved for future use
2..5	Double Integer	Pointer to a pixel image
6..7	Integer	Width
8..15	*Rect*	Bounds rectangle

Mask

A Mask is a data structure that determines how the pixels of an image are actually displayed. Only the pixels, in the desired display image, that correspond to ON bits (equal to 1) in the mask are drawn. If the mask has all its bits set to 1, then the entire original image is drawn. A mask is simply an array of integers where each element actually represents a bit pattern.

```
DIM aMask%(3)
```

PaintParam

The PaintParam data structure is only used in the PaintPixels Quickdraw routine. This data structure holds information pertinent to transfering a region of pixels without referencing the current GrafPort.

```
DIM aPaintParam!(21)
```

Element(s)	Value	Description
0..3	Double Integer	Pointer to source location information
4..7	Double Integer	Pointer to destination location information
8..11	Double Integer	Pointer to the source rectangle
12..15	Double Integer	Pointer to destination rectangle
16..17	Integer	Mode
18..21	Double Integer	Mask handle (ClipRgn)

Pattern

A pattern is an array of integers that represent bit patterns used by the Quickdraw pen for drawing. The pen pattern is pixel aligned so that it forms a continuous pattern in the areas it appears on the screen. Patterns on the Apple IIGS have "chunkiness" which means that each pixel is associated with 2 or 4 color bits in the pattern depending on the current graphics mode. A Pattern is defined as an array of 16 integers.

```
DIM aPattern%(15)
```

PenState

The PenState data structure is used for manipulating the current state of the Quickdraw pen. This is useful for routines that want to change the pen state briefly and then restore the previous pen values.

```
DIM aPenState!(47)
```

Element(s)	Value	Description
0..3	*Point*	Pen location
4..7	*Point*	Pen size
8..39	*Pattern*	Pen pattern
40..47	*Mask*	Pen mask

Picture (Pic)

The Pic data structure is a private data structure which defines a QuickDraw *picture*. A picture is a graphical object made up of one or more of the primitive Quickdraw objects (lines, rectangles, ovals, etc). The picture data structure should only be manipulated using the appropriate QuickDraw routines.

Point

The point data structure defines a location in the Quickdraw two-dimensional drawing space.

```
DIM aPoint%(1)
```

Element(s)	Value	Description
0	Integer	Horizontal coordinate
1	Integer	Vertical coordinate

Polygon (Poly)

The Poly data structure is a private data structure which defines a QuickDraw *polygon*. A polygon is a graphical object made up of one or more connected lines that together form a closed shape. The polygon data structure should only be manipulated using the appropriate QuickDraw routines.

QDProcs

It is possible to customize Quickdraw using the the QDProcs data structure. The QDProcs data structure consists of pointers to low level routines that other Quickdraw routines will call to accomplish their particular task. For example, FrameRect, PaintRect, FillRect, InvertRect all at some point call the same low level routine to draw the rectangle.

```
DIM aQDProcs@(12)
```

Element(s)	Value	Description
0	Double Integer	Pointer to text drawing procedure
1	Double Integer	Pointer to line drawing procedure
2	Double Integer	Pointer to rectangle drawing procedure
3	Double Integer	Pointer to round rectangle drawing procedure
4	Double Integer	Pointer to oval drawing procedure
5	Double Integer	Pointer to arc/wedge drawing procedure
6	Double Integer	Pointer to polygon drawing procedure
7	Double Integer	Pointer to region drawing procedure
8	Double Integer	Pointer to bit transfer procedure
9	Double Integer	Pointer to picture comment processing procedure
10	Double Integer	Pointer to text width measurement
11	Double Integer	Pointer to picture retrieval
12	Double Integer	Pointer to picture saving procedure

Rectangle (Rect)

The Rect data structure defines a rectangle given the top, left coordinate and bottom, right coordinate of the rectangle's corners.

```
DIM aRect%(3)
```

Element(s)	Value	Description
0	Integer	Top coordinate
1	Integer	Left coordinate
2	Integer	Bottom coordinate
3	Integer	Right coordinate

Region (Rgn)

The Rgn data structure is a private data structure which defines a QuickDraw *region*. A region is an arbitrary area or set of areas on the QuickDraw drawing space. The outline of a region must be one or more closed loops. The region data structure should only be manipulated using the appropriate QuickDraw routines.

SCB

The SCB or scanline control byte determines the pixel width and color palette for a specific horizontal line. Quickdraw also defines a Master SCB that is used by routines like InitPort to decide what standard values should initially be assigned into a GrafPort.

String

The string is a sequence of zero or more ASCII characters. The string also contains a *length byte* which stores the current number of characters in the string. This byte precedes the acutal characters in the string. The string data structure is sometimes called a *counted string* or a *Pascal string*. TML BASIC automatically converts BASIC strings from its string pool into strings which can be used with the Toolbox.

Routines

Housekeeping

```
DEF PROC QDStartUp(DPageAddr%,MasterSCB%,MaxWidth%,UserID%)     TOOL 2,4
DEF PROC QDShutDown                                              TOOL 3,4
DEF FN   QDVersion%                                              TOOL 4,4
DEF PROC QDReset                                                 TOOL 5,4
DEF FN   QDStatus%                                               TOOL 6,4
```

Global Environment

```
DEF FN   GetStandardSCB%                                         TOOL 12,4
DEF PROC SetMasterSCB(MasterSCB%)                                TOOL 22,4
DEF FN   GetMasterSCB%                                           TOOL 23,4
DEF PROC InitColorTable(ColorTablePtr@)                          TOOL 13,4
DEF PROC SetColorTable(TableNumber%,SrcColorTablePtr@)           TOOL 14,4
DEF PROC GetColorTable(TableNumber%,DestColorTablePtr@)          TOOL 15,4
DEF PROC SetColorEntry(TableNumber%,EntryNumber%,NewColor%)      TOOL 16,4
DEF FN   GetColorEntry%(TableNumber%,EntryNumber%)               TOOL 17,4
DEF PROC SetSCB(ScanLine%,NewSCB%)                               TOOL 18,4
DEF FN   GetSCB%(ScanLine%)                                      TOOL 19,4
DEF PROC SetAllSCBs(NewSCB%)                                     TOOL 20,4
DEF PROC SetSysFont(FontHndl@)                                   TOOL 178,4
DEF FN   GetSysFont@                                             TOOL 179,4
DEF PROC ClearScreen(Color%)                                     TOOL 21,4
DEF PROC GrafOn                                                  TOOL 10,4
DEF PROC GrafOff                                                 TOOL 11,4
```

GrafPort Manipulation

```
DEF PROC OpenPort(GrafPortPtr@)                                  TOOL 24,4
DEF PROC InitPort(GrafPortPtr@)                                  TOOL 25,4
DEF PROC ClosePort(GrafPortPtr@)                                 TOOL 26,4
DEF PROC SetPort(GrafPortPtr@)                              TOOL 27,4
DEF FN   GetPort@                                           TOOL 28,4
DEF PROC SetPortLoc(LocInfoRecPtr@)                         TOOL 29,4
DEF PROC GetPortLoc(LocInfoRecPtr@)                         TOOL 30,4
DEF PROC SetPortRect(RectPtr@)                              TOOL 31,4
DEF PROC GetPortRect(RectPtr@)                              TOOL 32,4
DEF PROC SetPortSize(Width%,Height%)                        TOOL 33,4
DEF PROC MovePortTo(H%,V%)                                  TOOL 34,4
DEF PROC SetOrigin(H%,V%)                                   TOOL 35,4
DEF PROC SetClip(RgnHndl@)                                  TOOL 36,4
DEF PROC GetClip(RgnHndl@)                                  TOOL 37,4
DEF PROC ClipRect(RectPtr@)                                 TOOL 38,4
```

Pen, Pattern, and Drawing

```
DEF PROC HidePen                                            TOOL 39,4
DEF PROC ShowPen                                            TOOL 40,4
DEF PROC GetPen(PointPtr@)                                  TOOL 41,4
DEF PROC SetPenState(PenStatePtr@)                          TOOL 42,4
DEF PROC GetPenState(PenStatePtr@)                          TOOL 43,4
DEF PROC SetPenSize(Width%,Height%)                         TOOL 44,4
DEF PROC GetPenSize(PointPtr)                               TOOL 45,4
DEF PROC SetPenMode(PenMode%)                               TOOL 46,4
DEF FN   GetPenMode%                                        TOOL 47,4
DEF PROC SetPenPat(PatternPtr@)                             TOOL 48,4
DEF PROC GetPenPat(PatternPtr@)                             TOOL 49,4
DEF PROC SetSolidPenPat(ColorNum%)                          TOOL 55,4
DEF PROC SetPenMask(MaskPtr@)                               TOOL 50,4
DEF PROC GetPenMask(MaskPtr@)                               TOOL 51,4
DEF PROC SetBackPat(PatternPtr@)                            TOOL 52,4
DEF PROC GetBackPat(PatternPtr@)                            TOOL 53,4
DEF PROC SetSolidBackPat(ColorNum%)                         TOOL 56,4
DEF PROC SolidPattern(PatternPtr@,ColorNum%)                TOOL 57,4
DEF PROC PenNormal                                          TOOL 54,4
DEF PROC MoveTo(H%,V%)                                      TOOL 58,4
DEF PROC Move(DH%,DV%)                                      TOOL 59,4

DEF PROC SetFont(NewFontHndl@)                              TOOL 148,4
DEF FN   GetFont@                                           TOOL 149,4
DEF PROC SetFontID(FontIDHndl@)                             TOOL 208,4
DEF FN   GetFontID@                                         TOOL 209,4
DEF PROC GetFontInfo(FontInfoRecHndl)                       TOOL 150,4
DEF FN   GetFGSize%                                         TOOL 207,4
DEF PROC GetFontGlobals(FontGlobalsRec@)                    TOOL 151,4
DEF PROC SetFontFlags(Flags%)                               TOOL 152,4
DEF FN   GetFontFlags%                                      TOOL 153,4

DEF PROC SetTextFace(TextFace%)                             TOOL 154,4
DEF FN   GetTextFace%                                       TOOL 155,4
DEF PROC SetTextMode(TextMode%)                             TOOL 156,4
DEF FN   GetTextMode%                                       TOOL 157,4
DEF PROC SetSpaceExtra(SpaceExtraPtr@)                      TOOL 158,4
DEF FN   SpaceExtraPtr@                                     TOOL 159,4
DEF PROC  SetTextSize(Size%)                                            TOOL 210,4
DEF FN    GetTextSize%                                                  TOOL 211,4

DEF PROC  SetCharExtra(ChExtraPtr@)                                     TOOL 212,4
DEF FN    GetCharExtra@                                                 TOOL 213,4
DEF PROC  SetForeColor(ForeColor%)                                      TOOL 160,4
DEF FN    GetForeColor%                                                 TOOL 161,4
DEF PROC  SetBackColor(BackColor%)                                      TOOL 162,4
DEF FN    GetBackColor%                                                 TOOL 163,4
DEF PROC  SetBufDims(MaxWidth%,MaxFontHeight%,MaxFBRExtent%)            TOOL 203,4
DEF PROC  ForceBufDims(MaxWidth%,MaxFontHeight%,MaxFBRExtent%)          TOOL 204,4
DEF PROC  SaveBufDims(SizeInfoPtr@)                                     TOOL 205,4
DEF PROC  RestoreBufDims(MaxWidth%,MaxFontHeight%,MaxFBRExtent%)        TOOL 206,4
DEF PROC  SetClipHandle(ClipRgnHndl@)                                   TOOL 198,4
DEF FN    GetClipHandle@                                                TOOL 199,4
DEF PROC  SetVisRgn(VisRgnHndl@)                                        TOOL 180,4
DEF PROC  GetVisRgn(VisRgnHndl@)                                        TOOL 181,4
DEF PROC  SetVisHandle(VisRgnHndl@)                                     TOOL 200,4
DEF PROC  GetVisHandle(VisRgnHndl@)                                     TOOL 201,4
DEF FN    GetPicSave@                                                   TOOL 63,4
DEF FN    GetRgnSave@                                                   TOOL 65,4
DEF FN    GetPolySave@                                                  TOOL 67,4

DEF PROC  SetGrafProcs(GrafProcsPtr@)                                   TOOL 68,4
DEF FN    GetGrafProcsPtr@                                              TOOL 69,4
DEF PROC  SetUserField(UserfieldPtr@)                                   TOOL 70,4
DEF FN    GetUserFieldPtr@                                              TOOL 71,4
DEF FN    GetSysFieldPtr@                                               TOOL 73,4
```

Drawing Lines

```
DEF PROC  LineTo(H%,V%)                                                 TOOL 60,4
DEF PROC  Line(DH%,DV%)                                                 TOOL 61,4
```

Drawing Rectangles

```
DEF PROC  FrameRect(RectPtr@)                                           TOOL 83,4
DEF PROC  PaintRect(RectPtr@)                                           TOOL 84,4
DEF PROC  EraseRect(RectPtr@)                                           TOOL 85,4
DEF PROC  InvertRect(RectPtr@)                                          TOOL 86,4
DEF PROC  FillRect(RectPtr@,PatternPtr@)                                TOOL 87,4
```

Drawing Regions

```
DEF PROC  FrameRgn(RgnHndl@)                                            TOOL 121,4
DEF PROC  PaintRgn(RgnHndl@)                                            TOOL 122,4
DEF PROC  EraseRgn(RgnHndl@)                                            TOOL 123,4
DEF PROC  InvertRgn(RgnHndl@)                                           TOOL 124,4
DEF PROC  FillRgn(RgnHndl@,PatternPtr@)                                 TOOL 125,4
```

Drawing Polygons

```
DEF PROC FramePoly(PolyHndl@)                                  TOOL 188,4
DEF PROC PaintPoly(PolyHndl@)                                  TOOL 189,4
DEF PROC ErasePoly(PolyHndl@)                                  TOOL 190,4
DEF PROC InvertPoly(PolyHndl@)                                 TOOL 191,4
DEF PROC FillPoly(PolyHndl@,PatternPtr@)                       TOOL 192,4
```

Drawing Ovals

```
DEF PROC FrameOval(RectPtr@)                                   TOOL 88,4
DEF PROC PaintOval(RectPtr@)                                   TOOL 89,4
DEF PROC EraseOval(RectPtr@)                                   TOOL 90,4
DEF PROC InvertOval(RectPtr@)                                  TOOL 91,4
DEF PROC FillOval(RectPtr@,PatternPtr@)                        TOOL 92,4
```

Drawing RoundRect s

```
DEF PROC FrameRRect(RectPtr@,OvalWidtH%,OvalHeight%)           TOOL 93,4
DEF PROC PaintRRect(RectPtr@,OvalWidth%,OvalHeight%)           TOOL 94,4
DEF PROC EraseRRect(RectPtr@,OvalWidth%,OvalHeight%)           TOOL 95,4
DEF PROC InvertRRect(RectPtr@,OvalWidth%,OvalHeight%)          TOOL 96,4
DEF PROC FillRRect(RectPtr@,OvalWidth%,OvalHeight%,PatternPtr@) TOOL 97,4
```

Drawing Arcs

```
DEF PROC FrameArc(RectPtr@,StartAngle%,ArcAngle%)              TOOL 98,4
DEF PROC PaintArc(RectPtr@,StartAngle%,ArcAngle%)              TOOL 99,4
DEF PROC EraseArc(RectPtr@,StartAngle%,ArcAngle%)              TOOL 100,4
DEF PROC InvertArc(RectPtr@,StartAngle%,ArcAngle%)             TOOL 101,4
DEF PROC FillArc(RectPtr@,StartAngle%,ArcAngle%,PatternPtr@)   TOOL 102,4
```

Pixel Transfer

```
DEF PROC ScrollRect(DstRectPtr@, DH%,DV%,UpdateRgnHndl@)       TOOL 126,4
DEF PROC PaintPixels(PaintParamPtr@)                           TOOL 127,4
DEF PROC PPToPort
            (SrcLocInfoRecPtr@,
             SrcRectPtr@,
             DestX%,
             DestY%,mode@)                                     TOOL 214,4
```

Text Drawing and Measuring

```
DEF PROC DrawChar(Char%)                                       TOOL 164,4
DEF PROC DrawText(TextPtr@,TextLength%)                        TOOL 167,4
DEF PROC DrawString(String$)                                   TOOL 165,4
DEF PROC DrawCString(CStringPtr@)                              TOOL 166,4

DEF FN   CharWidth%(Char%)                                     TOOL 168,4
DEF FN   TextWidth%(TextPtr@,TextLength%)                      TOOL 171,4
DEF FN   StringWidth%(String$)                                 TOOL 169,4
DEF FN   CStringWidth%(CStringPtr@)                            TOOL 170,4
DEF PROC CharBounds(Char%,RectPtr@)                                TOOL 172,4
DEF PROC TextBounds(TextPtr@,TextLength%, RectPtr@)                TOOL 175,4
DEF PROC StringBounds(String$,RectPtr@)                            TOOL 173,4
DEF PROC CStringBounds(StringPtr@,RectPtr@)                        TOOL 174,4
```

Calculations with Rectangles

```
DEF PROC SetRect(RectPtr@,Left%,Top%,Right%,Bottom%)               TOOL 74,4
DEF PROC OffsetRect(RectPtr@,DH%,DV%)                              TOOL 75,4
DEF PROC InsetRect(RectPtr@,DH%,DV%)                               TOOL 76,4
DEF FN   SectRect%(Src1RectPtr@, Src2RectPtr@,DstRectPtr@)         TOOL 77,4
DEF PROC UnionRect(Src1RectPtr@,Src2RectPtr@,DstRectPtr@)          TOOL 78,4
DEF FN   PtInRect%(PointPtr@,RectPtr@)                             TOOL 79,4
DEF PROC Pt2Rect(Point1Ptr@,Point2Ptr@,DstRectPtr@)                TOOL 80,4
DEF FN   EqualRect%(Rect1Ptr@,Rect2Ptr@)                           TOOL 81,4
DEF FN   EmptyRect%(RectPtr@)                                      TOOL 82,4
```

Calculations with Points

```
DEF PROC AddPt(SrcPointPtr@,DstPointPtr@)                          TOOL 128,4
DEF PROC SubPt(SrcPointPtr@,DstPointPtr@)                          TOOL 129,4
DEF PROC SetPt(PointPtr@,H%,V%)                                    TOOL 130,4
DEF FN   EqualPt%(Point1Ptr@,Point2Ptr@)                           TOOL 131,4
DEF PROC LocalToGlobal(PointPtr@)                                  TOOL 132,4
DEF PROC GlobalToLocal(PointPtr@)                                  TOOL 133,4
```

Calculations with Regions

```
DEF FN   NewRgn@                                                   TOOL 103,4
DEF PROC DisposeRgn(RgnHndl@)                                      TOOL 104,4
DEF PROC CopyRgn(SrcRgnHndl@,DstRgnHndl@)                          TOOL 105,4
DEF PROC SetEmptyRgn(RgnHndl@)                                     TOOL 106,4
DEF PROC SetRectRgn(RgnHndl@,Left%,Top%,Right%,Bottom%)            TOOL 107,4
DEF PROC RectRgn(RgnHandle,RectPtr@)                               TOOL 108,4
DEF PROC OpenRgn                                                   TOOL 109,4
DEF PROC CloseRgn(DstRgnHndl@)                                     TOOL 110,4
DEF PROC OffsetRgn(RgnHndl@,DH%,DV%)                               TOOL 111,4
DEF PROC InsetRgn(RgnHndl@,DH%,DV%)                                TOOL 112,4
DEF PROC SectRgn(SrcRgn1Hndl@,SrcRgn2Hndl@,DstRgnHndl@)            TOOL 113,4
DEF PROC UnionRgn(SrcRgn1Hndl@,SrcRgn2Hndl@,DstRgnHndl@)           TOOL 114,4
DEF PROC DiffRgn(SrcRgn1Hndl@,SrcRgn2Hndl@,DstRgnHndl@)            TOOL 115,4
DEF PROC XorRgn(SrcRgn1Hndl@,SrcRgn2Hndl@,DstRgnHndl@)             TOOL 116,4
DEF FN   PtInRgn%(PointPtr@,RgnHndl@)                              TOOL 117,4
DEF FN   RectInRgn%(RectPtr@,RgnHndl@)                             TOOL 118,4
DEF FN   EqualRgn%(Rgn1Handle@,Rgn2Hndl@)                          TOOL 119,4
DEF FN   EmptyRgn%(RgnHndl@)                                       TOOL 120,4
```

Calculations with Polygons

```
DEF FN   OpenPoly@                                                 TOOL 193,4
DEF PROC ClosePoly                                                 TOOL 194,4
DEF PROC KillPoly(PolyHndl@)                                       TOOL 195,4
DEF PROC OffsetPoly(PolyHndl@,DH%,DV%)                                TOOL 196,4
```

Operations with Pictures

```
DEF FN   OpenPicture@(picFrameRectPtr@)                               TOOL 183,4
DEF PROC PicComment(kind%,dataSize%,dataHndl@)                        TOOL 184,4
DEF PROC ClosePicture                                                 TOOL 185,4
DEF PROC DrawPicture(myPictureHndl@,dstRectPtr@)                      TOOL 186,4
DEF PROC KillPicture(myPictureHndl@)                                  TOOL 187,4
```

Mapping and Scaling Utilities

```
DEF PROC MapPt(PointPtr@,fromSrcRectPtr@,toDestRectPtr@)              TOOL 138,4
DEF PROC MapRect(RectPtr@,fromSrcRectPtr@,toDestRectPtr@)             TOOL 139,4
DEF PROC MapRgn(RgnHndl@,fromSrcRectPtr@,toDestRectPtr@)              TOOL 140,4
DEF PROC MapPoly(PolyHndl@,fromSrcRectPtr@,toDestRectPtr@)            TOOL 197,4
DEF PROC ScalePt(PointPtr@,fromSrcRectPtr@,toDestRectPtr@)            TOOL 137,4
```

Miscellaneous

```
DEF FN   Random%                                                      TOOL 134,4
DEF PROC SetRandSeed(Seed@)                                           TOOL 135,4
DEF FN   GetPixel%(H%,V%)                                             TOOL 136,4
```

Customizing QuickDraw

```
DEF PROC SetStdProcs(QDProcsPtr@)                                     TOOL 141,4
DEF FN   GetAddress%                                                  TOOL 9,4
```

Cursor-Handling

```
DEF PROC SetCursor(CursorPtr@)                                        TOOL 142,4
DEF FN   GetCursorAdr@                                                TOOL 143,4
DEF PROC HideCursor                                                   TOOL 144,4
DEF PROC ShowCursor                                                   TOOL 145,4
DEF PROC ObscureCursor                                                TOOL 146,4
DEF PROC InitCursor                                                   TOOL 202,4

DEF PROC InflateTextBuffer(NewWidth%,NewHeight%)                      TOOL 215,4
DEF PROC GetROMFont(ROMFontInfoRecPtr@)                               TOOL 216,4
DEF FN   GetFontLore(GlobalsPtr@,GlobalsSz%)                          TOOL 217,4
```

QuickDraw Auxiliary

The QuickDraw Auxiliary tool set contains additional graphics routines which complement the QuickDraw tool set. The QuickDraw Auxiliary routines are needed to use the Print Manager.

Special Values

No special values defined for Quickdraw Auxiliary.

Data Structures

No data structures defined for Quickdraw Auxiliary.

Routines

Housekeeping

```
DEF PROC QDAuxStartUp                                TOOL 2,18
DEF PROC QDAuxShutDown                               TOOL 3,18
DEF FN   QDAuxVersion%                               TOOL 4,18
DEF PROC QDAuxReset                                  TOOL 5,18
DEF FN   QDAuxStatus%                                TOOL 6,18
```

Miscellaneous

```
DEF PROC CopyPixels
              (SrcLocInfoRecPtr@,
               DstLocInfoRecPtr@,
               SrcRectPtr@,
               DstRectPtr@,
               Mode%,
               ClipRgnHndl@)                         TOOL 9,18
DEF PROC WaitCursor                                  TOOL 10,18
DEF PROC DrawIcon(IconPtr@,DisplayMode%,XPos%,YPos%) TOOL 11,18
```

Scheduler

The Scheduler tool set is responsible for delaying the activation of system tasks and desk accessories until the resources that the task/desk accessory requires become available. This library is necessary for writing applications that perform interrupt handling that access ProDOS 16 or the tool set routines. An example of an interrupt handler is an application that performs background printing. This library provides access to the Scheduler's low level system operations.

Special Values

No special values defined for the Scheduler.

Data Structures

No data structures defined for the Scheduler.

Routines

HouseKeeping

```
DEF PROC  SchStartUp                                    TOOL 2,7
DEF PROC  SchShutDown                                   TOOL 3,7
DEF FN    SchVersion%                                   TOOL 4,7
DEF PROC  SchReset                                      TOOL 5,7
DEF FN    SchStatus%                                    TOOL 6,7
```

Queue Access

```
DEF FN    SchAddTask%(TaskProcPtr@)                     TOOL 9,7
DEF PROC  SchFlush                                      TOOL 10,7
```

Scrap Manager

The Scrap Manager allows the transfering of data between an application and a data storage area called the clipboard. These routines are used to implement the standard Cut, Copy and Paste options found in the conventional Edit Menu.

Special Values

No special values defined for the Scrap Manager.

Data Structures

No data structures defined for the Scrap Manager.

Routines

HouseKeeping

```
DEF PROC  ScrapStartUp                                          TOOL 2,22
DEF PROC  ScrapShutDown                                         TOOL 3,22
DEF FN    ScrapVersion%                                         TOOL 4,22
DEF PROC  ScrapReset                                            TOOL 5,22
DEF FN    ScrapStatus%                                          TOOL 6,22
```

Miscellaneous

```
DEF PROC  UnLoadScrap                                           TOOL 9,22
DEF PROC  LoadScrap                                             TOOL 10,22
DEF PROC  ZeroScrap                                             TOOL 11,22
DEF PROC  PutScrap(NumBytes@,ScrapType%,SrcPtr@)                TOOL 12,22
DEF PROC  GetScrap(DestHndl@,ScrapType%)                        TOOL 13,22
DEF FN    GetScrapCount%                                        TOOL 18,22
DEF FN    GetScrapState%                                        TOOL 19,22
DEF FN    GetScrapHandle@(ScrapType%)                           TOOL 14,22
DEF FN    GetScrapSize@(ScrapType%)                             TOOL 15,22
DEF FN    GetScrapPath@                                         TOOL 16,22
DEF PROC  SetScrapPath(PathString$)                             TOOL 17,22
```

Sound Manager

The Sound Manager allows access to the Sound hardware without knowledge of the specific hardware I/O addresses. Since the Sound Manager routines can create basic sounds, other tool sets use it to create more complex sounds.

Special Values

No special values defined for the Sound Manager.

Data Structures

ParamBlkSoundRec

The ParamBlkSoundRec data structure contains all the necessary information that defines a sound for the Sound Manager to pass to the Ensoniq Digital Oscillator Chip (DOC). The frequency of the waveform playback in structure elements 5 and 6 can be calculated as follows:

FREQUENCY = ((32 * Playback Frequency in hertz) / 1645)

Waveforms are further defined in the Note Synthesizer library.

```
DIM ParamBlkSoundRec!(13)
```

Element(s)	Value	Description
0..1	Integer	Starting address of waveform
2	Integer	Starting bank of waveform
3..4	Integer	Size of waveform in pages, from 1 to $FFFF
5..6	Integer	Frequency of waveform playback
7..8	Integer	Starting address of DOC buffer
9	Integer	Code to specify size of DOC buffer
10..11	Integer	Starting address of next waveform, 0 if last waveform
12	Integer	Starting bank of next waveform, 0 if last waveform
13	Integer	Volume setting of waveform

Routines

Housekeeping

```
DEF PROC  SoundStartUp(DPageAddr%)                                TOOL 2,8
DEF PROC  SoundShutDown                                           TOOL 3,8
DEF FN    SoundVersion%                                           TOOL 4,8
DEF PROC  SoundReset                                              TOOL 5,8
DEF FN    SoundStatus%
```

RAM and Volume

```
DEF PROC   WriteRamBlock(SrcPtr@,DOCstart%,ByteCount%)              TOOL 9,8
DEF PROC   ReadRamBlock(DstPtr@,DOCstart%,ByteCount%)               TOOL 10,8
DEF FN     GetTableAddress@                                         TOOL 11,8
DEF FN     GetSoundVolume%(Generator%)                              TOOL 12,8
DEF PROC   SetSoundVolume(Volume%,Generator%)                       TOOL 13,8
```

Free-Form Synthesizer

```
DEF PROC   FFStartSound(DOCGenMode%,ParamBlkSoundRecPtr@)           TOOL 14,8
DEF PROC   FFStopSound(Generators%)                                 TOOL 15,8
DEF FN     FFSoundStatus%                                           TOOL 16,8
DEF FN     FFGeneratorStatus%(Generator%)                           TOOL 17,8
DEF PROC   SetSoundMIRQV(IRQVProcPtr@)                              TOOL 18,8
DEF FN     SetUserSoundIRQV@(newUserIRQVProcPtr@)                   TOOL 19,8
DEF FN     FFSoundDoneStatus%(Generator%)                           TOOL 20,8
```

Standard File

The Standard File Operations Tool Set provides a standard user interface for opening and saving files by supplying standard dialog boxes and routines to manipulate them.

Special Values

No special values defined for Standard File.

Data Structures

TypeList

The TypeList data structure defines the set of file types which a Standard File operation uses to determine which types of files to display. If the file type is set to 0, all file types will be displayed.

```
DIM aTypeList!(8)
```

Element(s)	Value	Description
0	Integer	Number of File Type entries in structure
1	Integer	File type 1
2	Integer	File type 2
3	Integer	File type 3
4	Integer	File type 4
5	Integer	File type 5
6	Integer	File type 6
7	Integer	File type 7
8	Integer	File type 8

ReplyRecord

The ReplyRecord data structure defines the information returned by the Standard File operations to indicate which file has been chosen. If element 0 contains a zero (0) then the remaining elements have no meaning.

```
DIM aReplyRecord!(149)
```

Element(s)	Value	Description
0	Integer	Non-zero if OPEN button pressed, 0 if CANCEL
1	Integer	Unused field
2..3	Integer	The filetype of the selected file
4..5	Integer 6	The AuxFileType of the selected file
6..31	*String!(~~15~~)*	The selected file's name stored as a counted string
32..159	*String!(~~127~~)* 22 [illegible]	The selected file's full pathname stored as a counted string

Routines

HouseKeeping

```
DEF PROC  SFStartUp(UserID%,DPageAddr%)                     TOOL 2,23
DEF PROC  SFShutDown                                        TOOL 3,23
DEF FN    SFVersion%                                        TOOL 4,23
DEF PROC  SFReset                                           TOOL 5,23
DEF FN    SFStatus%                                         TOOL 6,23
```

Standard Get and Put

```
DEF PROC  SFGetFile
              (WhereX%,
               WhereY%,
               PromptString$,
               FilterProcPtr@,
               TypeListPtr@,
               ReplyRecordPtr@)                             TOOL 9,23

DEF PROC  SFPutFile
              (WhereX%,
               WhereY%,
               PromptString$,
               OrigNameStringPtr@,
               MaxLen%,
               ReplyRecordPtr@)                             TOOL 10,23
```

Custom Get and Put

```
DEF PROC  SFPGetFile
             (WhereX%,
              WhereY%,
              PromptString$,
              FilterProcPtr@,
              TypeListPtr@,
              TheDialogPtr@,
              DialogHookProcPtr@,
              ReplyRecordPtr@)                                    TOOL 11,23

DEF PROC  SFPPutFile
             (WhereX%,
              WhereY%,
              PromptString$,
              OrigNameString$,
              MaxLen%,
              TheDialogPtr@,
              DialogHookProcPtr@,
              ReplyRecordPtr@)                                    TOOL 12,23

DEF PROC  SFAllCaps(AllCaps%)                                     TOOL 13,23
```

Text Tools

The Text Tools tool set provides an interface between Apple II character device drivers, which must be executed in emulation mode, and new applications running in native mode. The tool set also supports redirection of I/O to the Apple IIGS ports as well as dealing with the text screen without switching the 65816 processor modes.

Special Values

No special values for Text Tools.

Data Structures

No data structures for Text Tools.

Routines

Housekeeping

```
DEF PROC   TextStartUp                                         TOOL 2,12
DEF PROC   TextShutDown                                        TOOL 3,12
DEF FN     TextVersion%                                        TOOL 4,12
DEF PROC   TextReset                                           TOOL 5,12
DEF FN     TextStatus%                                         TOOL 6,12
```

Globals Manipulation

```
DEF PROC   SetInGlobals(ANDMask%, ORMask%)                     TOOL 9,12
DEF PROC   SetOutGlobals(ANDMask%,ORMask%)                     TOOL 10,12
DEF PROC   SetErrGlobals(ANDMask%, ORMask%)                    TOOL 11,12
DEF FN     GetInGlobals%[2]                                    TOOL 12,12
DEF FN     GetOutGlobals%[2]                                   TOOL 13,12
DEF FN     GetErrGlobals%[2]                                   TOOL 14,12
```

I/O Direction

```
DEF PROC   SetInputDevice(DeviceType%,SlotOrInitProcPtr@)      TOOL 15,12
DEF PROC   SetOutputDevice(DeviceType%,SlotOrInitProcPtr@)     TOOL 16,12
DEF PROC   SetErrorDevice(DeviceType%,SlotOrInitProcPtr@)      TOOL 17,12
DEF FN     GetInputDevice![6]                                  TOOL 18,12
DEF FN     GetOutputDevice![6]                                 TOOL 19,12
DEF FN     GetErrorDevice![6]                                  TOOL 20,12
```

Text I/O

```
DEF PROC   InitTextDev(DeviceNumber%)                          TOOL 21,12
DEF PROC   CtrlTextDev(DeviceNumber%,ControlCode%)             TOOL 22,12
DEF PROC   StatusTDev(DeviceNumber%,RequestCode%)              TOOL 23,12
DEF PROC   WriteChar(Char%)                                    TOOL 24,12
DEF PROC   ErrWriteChar(Char%)                                 TOOL 25,12
DEF PROC   WriteLine(String$)                                  TOOL 26,12
DEF PROC  ErrWriteLine(String$)                                TOOL 27,12
DEF PROC  WriteString(String$)                                 TOOL 28,12
DEF PROC  ErrWriteString(String$)                              TOOL 29,12
DEF PROC  TextWriteBlock(TextPtr@,Offset%ByteCount%)           TOOL 30,12
DEF PROC  ErrWriteBlock(TextPtr@,Offset%,ByteCount%)           TOOL 31,12
DEF PROC  WriteCString(CStringPtr@)                            TOOL 32,12
DEF PROC  ErrWriteCString(CStringPtr@)                         TOOL 33,12
DEF FN    ReadChar%(Echo%)                                     TOOL 34,12
DEF PROC  TextReadBlock(TextPtr@,Offset%,ByteCount%,Echo%)     TOOL 35,12
DEF FN    ReadLine%(TextPtr@,ByteCount%,EOLChar%,Echo%)        TOOL 36,12
```

Tool Locator

The Tool Locator is the most important of the Apple IIGS tool sets. The Tool Locator allows a program to load tool sets from disk into RAM and is responsible for locating a tool set routine when a program calls a Toolbox procedure or function.

Special Values

No special values defined for the Tool Locator.

Data Structures

ToolTable

The ToolTable is an array of integers used to describe which Apple IIGS Toolbox tool sets must be loaded to memory so that a program may use the requested toolset's routines. The size of the ToolTable structure can vary from application to application or even within the same application. The routines that use this structure determine the number of tools to load from the first element which also indirectly indicates the size of the structure.

Most TML BASIC programs will not use this data structure since the TML BASIC LIBRARY statement automatically generates a call to the Tool Locator LoadOneTool procedure in order to load the specified tool set.

```
DIM aToolTable%(N)
```

Element(s)	Value	Description
0	Integer	Specifies the number of tool sets specified in the table
1	Integer	Tool set number
2	Integer	Minimum version of tool set specified by element 1
3	Integer	Tool set number
4	Integer	Minimum version of tool set specified by element 3
5	Integer	Tool set number
6	Integer	Minimum version of tool set specified by element 5
•		
•		
•		
N	Integer	Minimum version of tool set specified by element (n-1)

Routines

Housekeeping

```
DEF PROC  TLStartUp                                           TOOL 2,1
DEF PROC  TLShutDown                                          TOOL 3,1
DEF FN    TLVersion%                                          TOOL 4,1
DEF PROC  TLReset                                             TOOL 5,1
DEF FN    TLStatus%                                           TOOL 6,1
```

Locator

```
DEF FN    GetTSPtr@(UserOrSystem%,TSNum%)                     TOOL 9,1
DEF PROC  SetTSPtr(UserOrSystem%,TSNum%,FPTptr@)              TOOL 10,1
DEF FN    GetFuncPtr@(UserOrSystem%,TSFuncNum%)               TOOL 11,1
DEF FN    GetWAP@(UserOrSystem%,TSNum%)                       TOOL 12,1
DEF PROC  SetWAP(UserOrSystem%,TSNum%,WAptr@)                 TOOL 13,1
DEF PROC  LoadTools(ToolTablePtr@)                            TOOL 14,1
DEF PROC  LoadOneTool(TSNum%,MinVersion%)                     TOOL 15,1
DEF PROC  UnLoadOneTool(TSNum%)                               TOOL 16,1

DEF FN    TLMountVolume%
             (WhereX%,
              WhereY%,
              Line1String$,
              Line2String$,
              Button1String$,
              Button2String$)                                 TOOL 17,1

DEF FN    TLTextMountVolume%
             (Line1String$,
              Line2String$,
              Button1String$,
              Button2String$)                                 TOOL 18,1
```

Window Manager

The Window Manager creates the desktop environment and is responsible for the create and manipulation of windows.

Special Values

The FindWindow% function result reports which part of which window, if any, a mouse button was pressed in.

Values	Description of Value
0	Mouse button pressed in desk area
1	Mouse button pressed in content area
2	Mouse button pressed in go-away area
3	Mouse button pressed in drag area

The GetWKind% function result returns an integer indicating which kind of window is the window specified by the passed parameter.

Values	Description of Value
-32768	Desk accessory window
others	Application window

Data Structures

Window

The Window data structure is a private data structure returned by the Window Manager function NewWindow@. The contents of the Window data structure are not defined since a program should never actually reference any component of the data structure, but use the various Window Manager routines instead.

WindowColorTbl

The WindowColorTbl data structure defines the color values for the different components of a window.

```
DIM aWindowColorTbl%(4)
```

Element(s)	Value	Description
0	Integer	Window's frame color
1	Integer	Window's title color
2	Integer	Window's titlebar color
3	Integer	Window's grow box color
4	Integer	Window's information bar color

NewWindowParamBlk

The NewWindowParamBlk data structure defines to the NewWindow@ function how to create a window. See Chapter 13 for complete information on how to use this data structure.

```
DIM aNewWindowParamBlk!(73)
```

Element(s)	Value	Description
0..1	Integer	Number of bytes in NewWindowParamBlk (=74)
2..3	Integer	Bit vector that describes the window
4..7	Double Integer	Pointer to window's title : *String*Ptr
8..11	Double Integer	Application RefCon
12..19	*Rect*	Size and position of content when zoomed
20..23	Double Integer	Pointer to window's color table : *WindowColorTbl*Ptr
24..25	Integer	Content's vertical origin
26..27	Integer	Content's horizontal origin
28..29	Integer	Entire height of document
30..31	Integer	Entire width of document
32..33	Integer	Maximum height of content allowed by GrowWindow
34..35	Integer	Maximum width of content allowed by GrowWindow
36..37	Integer	Number of pixels to scroll vertically for arrows
38..39	Integer	Number of pixels to scroll horizontally for arrows
40..41	Integer	Number of pixels to scroll vertically for page
42..43	Integer	Number of pixels to scroll horizontally for page
44..47	Double Integer	Information bar refcon
48..49	Integer	Height of information bar
50..53	Double Integer	Address of standard window definition procedure : *Proc*Ptr
54..57	Double Integer	Address of information bar procedure : *Proc*Ptr
58..61	Double Integer	Address of content update draw procedure : *Proc*Ptr
62..65	*Rect*	Starting position and size of window

66..69	Double Integer	Window's starting plane
70..73	Double Integer	Address of memory to use for window record

Routines

Housekeeping

```
DEF PROC   WindStartUp(UserID%)                                    TOOL 2,14
DEF PROC   WindShutDown                                            TOOL 3,14
DEF FN     WindVersion%                                            TOOL 4,14
DEF PROC   WindReset                                               TOOL 5,14
DEF FN     WindStatus%                                             TOOL 6,14
DEF PROC   WindNewRes                                              TOOL 37,14
```

Creating and Disposing

```
DEF FN     NewWindow@(NewWindowParamBlkPtr@)                       TOOL 9,14
DEF PROC   CloseWindow(WindowPtr@)                                 TOOL 11,14
```

Window Record and Global Access

```
DEF FN     GetWMgrPort@                                            TOOL 32,14
DEF FN     SetWindowIcons@(NewFontHndl@)                           TOOL 78,14
DEF PROC   SetWRefCon(RefCon@,WindowPtr@)                          TOOL 40,14
DEF FN     GetWRefCon@(WindowPtr@)                                 TOOL 41,14
DEF PROC   SetWTitle(TitleString$,WindowPtr@)                      TOOL 13,14
DEF FN     GetWTitle@(WindowPtr@)                                  TOOL 14,14
DEF PROC   SetFrameColor(WindowColorTblPtr@, WindowPtr@)           TOOL 15,14
DEF PROC   GetFrameColor(WindowColorTblPtr@,WindowPtr@)            TOOL 16,14
DEF FN     FrontWindow@                                            TOOL 21,14
DEF FN     GetNextWindow@(WindowPtr@)                              TOOL 42,14
DEF FN     GetWKind%(WindowPtr@)                                   TOOL 43,14
DEF FN     GetWFrame%(WindowPtr@)                                  TOOL 44,14
DEF PROC   SetWFrame(WFlag%,WindowPtr@)                            TOOL 45,14
DEF FN     GetStructRgn@(WindowPtr@)                               TOOL 46,14
DEF FN     GetContentRgn@(WindowPtr@)                              TOOL 47,14
DEF FN     GetUpdateRgn@(WindowPtr@)                               TOOL 48,14
DEF FN     GetDefProc@(WindowPtr@)                                 TOOL 49,14
DEF PROC   SetDefProc(WDefProcPtr@,WindowPtr@)                     TOOL 50,14
DEF FN     GetWControls@(WindowPtr@)                               TOOL 51,14
DEF FN     GetZoomRect@(WindowPtr@)                                TOOL 55,14
DEF PROC   SetZoomRect(WFullSizeRectPtr@,WindowPtr@)               TOOL 56,14
DEF FN     GetSysWFlag%(WindowPtr@)                                TOOL 76,14
DEF PROC   SetSysWindow(WindowPtr@)                                TOOL 75,14
DEF FN     GetContentOrigin@(WindowPtr@)                           TOOL 62,14
DEF PROC   SetContentOrigin(XOrigin%,YOrigin%,WindowPtr@)          TOOL 63,14
DEF PROC   SetOriginMask(OriginMask%,WindowPtr@)                   TOOL 52,14
DEF PROC   StartDrawing(WindowPtr@)                                TOOL 77,14
DEF FN     GetDataSize@(WindowPtr@)                                TOOL 64,14
DEF PROC   SetDataSize(DataWidth%,DataHeight%,WindowPtr@)          TOOL 65,14
DEF FN     GetMaxGrow@(WindowPtr@)                                 TOOL 66,14
DEF PROC  SetMaxGrow(MaxWidth%,MaxHeight%,WindowPtr@)           TOOL 67,14
DEF FN    GetScroll@(WindowPtr@)                                TOOL 68,14
DEF PROC  SetScroll(HScroll%,VScroll%,WindowPtr@)               TOOL 69,14
DEF FN    GetPage@(WindowPtr@)                                  TOOL 70,14
DEF PROC  SetPage(HPage%,VPage%,WindowPtr@)                     TOOL 71,14
DEF FN    GetContentDraw@(WindowPtr@)                           TOOL 72,14
DEF PROC  SetContentDraw(ContDrawProcPtr@,WindowPtr@)           TOOL 73,14
```

Information Bar Access

```
DEF FN    GetInfoDraw@(WindowPtr@)                              TOOL 74,14
DEF PROC  SetInfoDraw(InfoDrawProcPtr@,WindowPtr@)              TOOL 22,14
DEF FN    GetInfoRefCon@(WindowPtr@)                            TOOL 53,14
DEF PROC  SetInfoRefCon(InfoRefCon@,WindowPtr@)                 TOOL 54,14
DEF PROC  GetRectInfo(InfoRectPtr@,WindowPtr@)                  TOOL 79,14
DEF PROC  StartInfoDrawing(InfoRectPtr@,WindowPtr@)             TOOL 80,14
DEF PROC  EndInfoDrawing                                        TOOL 81,14
```

Window Shuffling

```
DEF PROC  SelectWindow(WindowPtr@)                              TOOL 17,14
DEF PROC  HideWindow(WindowPtr@)                                TOOL 18,14
DEF PROC  ShowWindow(WindowPtr@)                                TOOL 19,14
DEF PROC  ShowHide(ShowFlag%,WindowPtr@)                        TOOL 35,14
DEF PROC  BringToFront(WindowPtr@)                              TOOL 36,14
DEF PROC  SendBehind(BehindWindowPtr@,WindowPtr@)               TOOL 20,14
```

Window Drawing

```
DEF PROC  HiliteWindow(FHilite%,WindowPtr@)                     TOOL 34,14
DEF PROC  Refresh(ClobberedRectPtr@)                            TOOL 57,14
```

User Interaction

```
DEF FN    FindWindow%(WindowPtr@,PointX%,PointY%)               TOOL 23,14
DEF PROC  DragWindow
              (Grid%,
               StartX%,
               StartY%,
               Grace%,
               BoundsRectPtr@,
               WindowPtr@)                                      TOOL 26,14
DEF FN    GrowWindow@
              (MinWidth%,
               MinHeight%,
               StartX%,
               StartY%,
               WindowPtr@)                                      TOOL 27,14
DEF FN    TrackGoAway%(StartX%,StartY%,WindowPtr@)              TOOL 24,14
DEF FN    TrackZoom%(StartX%,StartY%,WindowPtr@)                TOOL 38,14
DEF FN    TaskMaster%(EventMask%,EventRecordPtr@)               TOOL 29,14
```

Sizing and Positioning

```
DEF PROC  MoveWindow(NewX%,NewY%,WindowPtr@)                    TOOL 25,14
DEF PROC  SizeWindow(NewWidth%,NewHeight%,WindowPtr@)           TOOL 28,14
DEF PROC  ZoomWindow(WindowPtr@)                                TOOL 39,14
```

Update Region Maintenance

```
DEF PROC  InvalRect(BadRectPtr@)                                TOOL 58,14
DEF PROC  InvalRgn(BadRgnHndl@)                                 TOOL 59,14
DEF PROC  ValidRect(GoodRectPtr@)                               TOOL 60,14
DEF PROC  ValidRgn(GoodRgnHndl@)                                TOOL 61,14
DEF PROC  BeginUpdate(WindowPtr@)                               TOOL 30,14
DEF PROC  EndUpdate(WindowPtr@)                                 TOOL 31,14
```

Miscellaneous

```
DEF FN    PinRect@(TheXPt%,TheYPt%,TheRectPtr@)                 TOOL 33,14
DEF FN    CheckUpdate%(EventRecordPtr@)                         TOOL 10,14
DEF FN    WindowGlobal%(ChgFlag%)                               TOOL 86,14
DEF FN    GetFirstWindow@                                       TOOL 82,14
```

Appendix D
Comparing TML BASIC with GS BASIC

This appendix is intended for programmers familiar with the GS BASIC interpreter from Apple Computer who wish to begin using TML BASIC. Since the foundation for the design of TML BASIC is actually GS BASIC, you should find it quite easy to begin programming with TML BASIC. In fact, most programs written in GS BASIC will compile in TML BASIC with little or no changes.

The differences between TML BASIC and GS BASIC can be grouped into three major categories:

compiler versus interpreter differences,

extensions to GS BASIC, and

TML BASIC compiler issues.

Each of the sections in this chapter addresses these major categories of differences between GS BASIC and TML BASIC. The final section in this appendix describes the GS BASIC statements that can be used to *export* programs out of GS BASIC so that they can be compiled with TML BASIC.

Compiler / Interpreter Differences

Of course, the most significant difference between these two products is that TML BASIC is a compiler and GS BASIC is an interpreter. Chapter 1 of this manual provides a good discussion of the fundamental differences between a compiler and an interpreter. If you do not understand the meanings of these two words, you should review that discussion.

One of the biggest differences between an interpreter and a compiler is that an interpreter tends to confuse the issue of which commands are used for *creating* programs and those actually *used* in a program. That is, it becomes confusing as to which commands are part of the development system (the interpreter) and which commands are part of the programming language. For example, the GS BASIC commands AUTO, EDIT and LIST are not really part of the programming language like IF, LET and PRINT.

TML BASIC provides a multi-window, mouse-based editing and development environment. As such, TML BASIC does not require commands for loading, saving and other operations related to the process of creating programs. These operations are available as menu commands in TML BASIC. Thus, TML BASIC does not support the commands shown in Table D-1 which are typically used in the GS BASIC immediate mode.

Table D-1
Unsupported GS BASIC Immediate Mode Commands

AUTO	LIST
DEL	LISTTAB
EDIT	LOAD
EDIT TO#	NEW
EXEC	OUTREC
HLIST	RENUM
INDENT	SAVE

Unsupported Statements and Functions

In addition to the commands described in the previous section, TML BASIC does not support several other statements and functions available in the GS BASIC language. These statements and functions are not available because they relate directly to the characteristics of the interpreter which are not appropriate for a compiled language. For example, the BASIC@ function returns information specific to the internals of the interpreter such as the author's name, special global variables, etc. Clearly this information is not applicable to TML BASIC.

The following paragraphs outline these unsupported statements and functions.

BASIC@	In GS BASIC, this function is used to provide access to various internal pieces of information related to GS BASIC. This information is not applicable to the compiled language TML BASIC, and therefore not implemented.
CONT	TML BASIC does not permit an interrupted program to be restarted. Interrupting a program with STOP or Control-C is the same as executing the END statement.
COPY	In TML BASIC, you can use the powerful Cut, Copy and Paste operations to copy code between different source files. You may also use the GS Finder to copy and move files.

DIR	In TML BASIC, the standard Open File dialog and Save File dialog are used. Using these dialogs, it is possible to list all the files available on the disk. A program which needs to list the contents of a directory to the screen can use the CATALOG statement.
INVOKE	TML BASIC does not support Invokable OMF object code files.
LIBFIND	TML BASIC resolves all references to Toolbox procedures and functions during compile time. There is no need to search for Toolbox names during execution.
NOTRACE	TML BASIC provides no means of tracing the execution of compiled programs.
PERFORM	See INVOKE above.
PROGNAM$	This function works with the SAVE command which is not supported in TML BASIC. A compiled and executing TML BASIC program should not need the name of its source code file.
QUIT	To leave the TML BASIC environment you choose the Quit command from the File menu.
TRACE	See NOTRACE above.

Statements Requiring Modification

Finally, there are a few statements and functions in GS BASIC which also exist in TML BASIC, but behave a bit differently and may require modifying your programs before they will operate properly under TML BASIC. Chapter 10 identifies the implementation differences between GS BASIC and TML BASIC for every statement, function and reserved variable where such a case exists. The following are the most significant of these.

CHAIN	The CHAIN statement may only transfer control to another compiled ProDOS 16 application. Further, it is not possible to specify a label for execution to begin in the chained program. Execution will begin at the start of the program. GS BASIC chains to other GS BASIC source code programs. Thus, to use CHAIN in TML BASIC the chained program must also be a compiled program.
CLEAR	The CLEAR statement in TML BASIC only resets global variables. The alternate forms of the CLEAR statement used to reset the size of

	the data segment, library segment or invokable module segments have no meaning in TML BASIC.
DIM	The DIM statement behaves quite differently in TML BASIC. In TML BASIC, the DIM statement is used to create static dimensioned arrays, and is processed at compile time rather than execution time. To create arrays with dynamic valued dimensions, the DIM DYNAMIC statement must be used. See Chapter 7 for a thorough discussion of the DIM and DIM DYNAMIC statements.
FRE	In GS BASIC, this function returns the amount of memory available in the interpreter's data segment. In TML BASIC, a "data segment" does not exist since global variable allocation is restricted only by available memory. Thus, this function returns the amount of free memory in the Apple IIGS.
FREEMEM	This function is used to return information about GS BASIC's memory utilization. Much of this information is specific to GS BASIC as an interpreter and thus, not applicable to TML BASIC. See Chapter 10 for a complete description of this function in TML BASIC.
RUN	This statement is used to quit the currently executing program and transfer control to another program. In TML BASIC, the next program must be a compiled ProDOS 16 application, while in GS BASIC, the next program is the GS BASIC source code for a program.
TASKPOLL	In TML BASIC this statement must be executed to determine if an event has occurred. In GS BASIC, the detection of an event is automatic. See Chapters 10 and 13 for more information about this statement.

Execution / Compilation Order of Programs

One of the most significant differences between GS BASIC and TML BASIC is the manner in which programs are *processed*. Other than a very crude level of syntax checking, GS BASIC does not examine the source code of a program until it is executed; and then only in the order in which it is executed. Thus, the meaning of a statement depends upon when it is executed. In fact, because a statement is re-processed when it is executed again, a statement may behave very differently when it is executed a second time.

For example, the following code fragment contains a "Type Mismatch Error", however it is not reported by the interpreter.

```
myVar% = 1
IF myVar% < 99 THEN PROC reCalc(myVar%)
ELSE myVar% = newVal$
```

Since the variable *myVar%* is less than 99, the THEN part of the IF statement is executed. Thus, the ELSE part is not executed, and the error in the ELSE statement goes undetected. TML BASIC, however, examines every line of code regardless of any particular execution order. TML BASIC does this because it must generate code for every statement in the program in order to create a stand-alone program. Therefore, it will report this error.

While this error does not seem too dangerous, since it will eventually be found when *myVar%* is greater than 99, other errors are not so obvious. For example, consider the following code fragment.

```
GOTO doDim

Initialize: Score(15) = 92 : Score(16) = 83 : Score(17) = 86
            GOTO CalcAvg

doDim:      DIM Score(20)
            GOTO Initialize
```

Because GS BASIC processes statements in execution order, the array *Score* is first dimensioned with 21 elements and then the statements which initialize the elements of the array are executed. In this case, the code fragment executes perfectly in GS BASIC, but such is not the case in TML BASIC. Because TML BASIC processes the source code one line after the other, from top to bottom, the statement *Score(15) = 92* is processed before the *DIM Score(20)* statement. Since the array has not yet been declared, TML BASIC implicitly declares the array, but with only 11 elements. Thus, the assignment statement causes a runtime error when the program is executed because the array element *Score(15)* does not exist. See Chapter 7 for more information about arrays.

Extensions to GS BASIC

TML BASIC has added two major extensions to the GS BASIC language. In addition, several existing GS BASIC statements and functions have increased functionality. The following two paragraphs highlight the significant extensions, however, you should reference Part II of this manual for a complete discussion of these and other TML BASIC features.

IF Block Statement

A significant enhancement to GS BASIC is the addition of the ELSEIF and END IF statements for creating multi-line structured if statements. This feature allows you to have several ELSE conditions in an IF statement, each of which may have several lines of code.

Libraries

TML BASIC allows you to construct programs out of one or more separately compiled libraries. A library allows you to divide your program into smaller more manageable chunks. Libraries also provide an excellent means of sharing code between different programs.

TML BASIC Compiler Issues

Finally, the nature of the TML BASIC implementation presents yet some other differences to consider. These issues are outlined in the following sections.

The TML BASIC Editor and Large Programs

The mouse-based editor integrated within TML BASIC does not currently allow the source code of programs to be larger than 32K bytes. While most BASIC programs should have no problem with this restriction, it is certain that some will. To help avoid this problem you should be sure to use a small INDENT and LISTTAB value when exporting a program out of GS BASIC (see section below on how to export files). This will reduce the number of extra spaces GS BASIC will generate to the text file it creates, thus reducing its file size.

For very large programs, no technique of squeezing out blanks spaces, comments, etc. will satisfy the 32K limit of the TML BASIC editor. In these cases, you must rely on separately compiled libraries. Each of one or more libraries will contain a portion of the large program. Study Chapter 8 for more information about libraries.

Segmentation

GS BASIC has a single code segment and data segment. The code segment contains the source code of a GS BASIC program, while the data segment stores global variables during the execution of a program. The size of these segments are controlled by the CLEAR statement.

TML BASIC on the other hand generates native 65816 machine code for programs. Thus, the compiled program must obey memory segmentation rules of the Apple IIGS for programs. The most significant rule is that code and data segments are limited to 64K bytes in size. If the code for a program grows larger than this, the program must be segmented into multiple code segments using the compiler's *$CSeg* metastatement. Likewise, if a program declares more than 64K bytes of global variables, the global variables must be segmented into multiple data segments using the *$DSeg* metastatement. See Appendix B for more information about segmenting programs.

Expression Evaluation

When TML BASIC generates code for an expression, it must decide at compile time the representation type to use for the evaluation. For example, if a program adds two *integer* variables together, TML BASIC generates code which evaluates the expression using 16-bit *integer* precision. If the result of the addition causes an overflow, an error is raised at execution time. For example, the code fragment shown here will encounter an overflow error when adding *x%* with 25000.

```
x% = 20000
y@ = x% + 25000
```

The execution of the second statement overflows in TML BASIC because the result of the addition, 45000, can not be represented as a 16-bit integer value. In order for this expression to evaluate properly the variable *x%* must first be converted to a double integer so that 32-bit double integer arithmetic is used. For example,

```
y@ = CONV@(x%) + 25000
```

In GS BASIC, an overflow error is not reported when adding x% with 25000 because, GS BASIC will, at execution time, convert both values to double integers and re-evaluate the expression. Whenever GS BASIC encounters an overflow error during execution, it converts both arguments to the next larger representation and re-evaluates the expression. This process is repeated until the expression evaluates properly, or until the largest representation is used and an error still occurs.

In this example, the evaluation of *x% + 25000* causes an overflow error when evaluated using integer arithmetic, so GS BASIC converts both arguments to double integer and re-evaluates the expression, this time without an error. Of course, if the variable y@ had been an integer variable, then GS BASIC would generate an "Overflow Error" on the assignment since the value 45000 could not be stored in the integer variable.

Exporting GS BASIC Programs into TML BASIC

GS BASIC stores the source code for programs in a special binary encoded format. Since TML BASIC stores the source code for programs as ASCII text files, a GS BASIC program must be converted to a text file before TML BASIC can compile it. To save a GS BASIC program as a text file requires only a few simple commands using GS BASIC.

First, the GS BASIC program to be converted must be loaded into memory. Then a text file which will contain the exported file must be created and opened. The standard output is then redirected to the open file and the source code listed to the file. Finally, the output file is closed. The following commands illustrate how to accomplish this process for the GS BASIC program named "GSBasicProg" and the export text file named "TMLBasicProg".

```
    ) LOAD GSBasicProg
    ) CREATE TMLBasicProg,FILTYP=TXT
    ) OPEN TMLBasicProg, AS #1
    ) OUTREC = 0: LISTTAB = 128
    ) OUTPUT #1: LIST: OUTPUT #0
    ) OUTREC = 80: LISTTAB = 5
    ) CLOSE #1
```

In this example, the OUTREC=0 statement is used to instruct GS BASIC not wrap the source code lines when listed. TML BASIC supports lines up to 255 characters. The LISTTAB=128 instructs GS BASIC not to list line numbers with the source code when the LIST statement is used. The OUTPUT #1 statement redirects output from the LIST statement to the file opened as #1. The LIST statement then lists the source code to the export file without line numbers and without wrapping long lines. Normal output is restored using the OUTPUT #0 statement. The standard settings for OUTREC and LISTTAB are restored and the export text file is closed.

Appendix E
ASCII Character Set

This appendix contains a complete list of the ASCII (American Standard Code for Information Interchange) character set. The characters whose values are greater than 127 are not actually part of the ASCII standard, but are the character definitions for Apple IIGS fonts, and so they are included here. While the character definitions given for values greater than 127 are not necessarily available in every font, they are the standard character definitions.

Decimal	Hex	Character	Names/Comments
0	00	Control-@	NUL, null
1	01	Control-A	
2	02	Control-B	
3	03	Control-C	Break
4	04	Control-D	
5	05	Control-E	
6	06	Control-F	
7	07	Control-G	BEL, bell
8	08	Control-H	BS, backspace
9	09	Control-I	HT, horizontal tab
10	0A	Control-J	LF, line feed
11	0B	Control-K	VT, vertical tab
12	0C	Control-L	FF, form feed, Page
13	0D	Control-M	CR, carriage return
14	0E	Control-N	
15	0F	Control-O	
16	10	Control-P	
17	11	Control-Q	XON, resume
18	12	Control-R	
19	13	Control-S	XOFF, screen pause
20	14	Control-T	
21	15	Control-V	
22	16	Control-U	
23	17	Control-W	
24	18	Control-X	CAN, cancel line
25	19	Control-Y	
26	1A	Control-Z	End of file

Decimal	Hex	Character	Names/Comments
27	1B	Control-[	ESC, escape
28	1C	Control-\	
29	1D	Control-]	
30	1E	Control-^	
31	1F	Control-_	
32	20		Space
33	21	!	Exclamation point
34	22	"	Quote
35	23	#	Pound sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Left parenthesis
41	29	)	Right parenthesis
42	2A	*	Asterisk
43	2B	+	Plus sign
44	2C	,	Comma
45	2D	-	Minus sign, dash
46	2E	.	Period
47	2F	\	Backlash
48	30	0	
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than
61	3D	=	Equal
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	At sign
65	41	A	

Decimal	Hex	Character	Names/Comments
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[	Left bracket
92	5C	\	Backlash
93	5D	]	Right bracket
94	5E	^	Caret
95	5F	_	Underscore
96	60	`	Accent grave
97	61	a	
98	62	b	
99	63	c	
100	64	d	
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	

Decimal	Hex	Character	Names/Comments
106	6A	j	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	Left brace
124	7C	|	Vertical line
125	7D	}	Right brace
126	7E	~	Tilde
127	7F	DEL	delete, rubout
128	80	Ä	
129	81	Å	
130	82	Ç	
131	83	É	
132	84	Ñ	
133	85	Ö	
134	86	Ü	
135	87	á	
136	88	à	
137	89	â	
138	8A	ä	
139	8B	ã	
140	8C	å	
141	8D	ç	
142	8E	é	
143	8F	è	
144	90	ê	
145	91	ë	

Decimal	Hex	Character	Names/Comments
146	92	í	
147	93	ì	
148	94	î	
148	95	ï	
150	96	ñ	
151	97	ó	
152	98	ò	
153	99	ô	
154	9A	ö	
155	9B	õ	
156	9C	ú	
157	9D	ù	
158	9E	û	
159	9F	ü	
160	A0	†	
161	A1	°	
162	A2	¢	
163	A3	£	
164	A4	§	
165	A5	•	
166	A6	¶	
167	A7	ß	
168	A8	®	
169	A9	©	
170	AA	™	
171	AB	´	
172	AC	¨	
173	AD	≠	
174	AE	Æ	
175	AF	Ø	
176	B0	∞	
177	B1	±	
178	B2	≤	
179	B3	≥	
180	B4	¥	
181	B5	µ	
182	B6	∂	
183	B7	Σ	
184	B8	Π	

Decimal	Hex	Character	Names/Comments
185	B9	π	
186	BA	∫	
187	BB	ª	
188	BC	º	
189	BD	Ω	
190	BE	æ	
191	BF	ø	
192	C0	¿	
193	C1	¡	
194	C2	¬	
195	C3	√	
196	C4	ƒ	
197	C5	≈	
198	C6	Δ	
299	C7	«	
200	C8	»	
201	C9	…	
202	CA		Nonbreaking space
203	CB	À	
204	CC	Ã	
205	CD	Õ	
206	CE	Œ	
207	CF	œ	
208	D0	–	
209	D1	—	
210	D2	“	
211	D3	”	
212	D4	‘	
213	D5	’	
214	D6	÷	
215	D7	◊	
216	D8	Ÿ	

Appendix F
ProDOS File Types

The following table contains a list of the most common ProDOS file types.

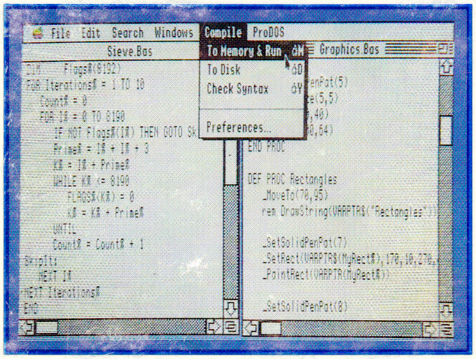
Mnemonic Code	File Type	Description
UNK	$00	Unknown
BAD	$01	Bad block file
TXT	$04	ASCII text file (SOS & ProDOS)
DIR	$0F	Subdirectory file (SOS & ProDOS)
ADB	$19	AppleWorks data base file
AWP	$1A	AppleWorks word processor file
ASP	$1B	AppleWorks spreadsheet file
GSB	$AB	IIGS BASIC program file
TDF	$AC	IIGS BASIC toolbox definition file
BDF	$AD	IIGS BASIC data file
SRC	$B0	APW text file
OBJ	$B1	APW object file
LIB	$B2	APW library file
S16	$B3	ProDOS 16 application file (OMF load)
RTL	$B4	APW run-time library file
EXE	$B5	APW shell application file
PPI	$B6	ProDOS 16 permanent Init file
PTI	$B7	ProDOS 16 temporary Init file
NDA	$B8	New desk accessory
CDA	$B9	Classic desk accessory
TOL	$BA	ProDOS 16 tool set file
DVR	$BB	ProDOS 16 driver file
$C0	$C0	Packed Super Hi-Res graphics file
PIC	$C1	Unpacked Super Hi-Res graphics file
WAV	$E1	IIGS BASIC Wavebank data file
PAS	$EF	Pascal area on a partitioned volume
OS	$F9	ProDOS 16 operating system
BAS	$FC	AppleSoft program file
VAR	$FD	AppleSoft variable file

Suggested Retail Price $125.00

TML BASIC for the Apple IIGS

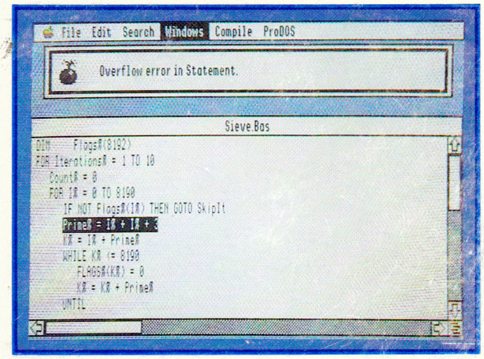
TML BASIC is a modern 16-bit, *compiled* implementation of the BASIC programming language designed to meet the needs of the broadest range of programmers possible for the Apple IIGS. TML BASIC operates within an elegant yet powerful programming environment that lets you write, edit, compile and run applications with incredible speed and simplicity.

Complete access to every Apple IIGS Toolbox routine is provided through predefined BASIC procedures and functions allowing the programmer to write applications implementing powerful IIGS Toolbox features with ease. TML BASIC also allows the programmer to write *textbook* programs which do not require any knowledge or use of the Toolbox. Compile and run BASIC programs directly to memory in seconds, or create stand-alone ProDOS 16 applications capable of running independently of TML BASIC and transferable to any Apple IIGS disk.



```
DIM     Flags%(8192)
FOR Iterations% = 1 TO 10
  Count% = 0
  FOR I% = 0 TO 8190
    IF NOT Flags%(I%) THEN GOTO Sk
    Prime% = I% + I% + 3
    K% = I% + Prime%
    WHILE K% <= 8190
      FLAGS%(K%) = 0
      K% = K% + Prime%
    UNTIL
    Count% = Count% + 1
SkipIt:
  NEXT I%
NEXT Iterations%
END
```

```
enPat(5)
e(5,5)
,40)
0,64)
END PROC

DEF PROC Rectangles
  _MoveTo(70,95)
  rem DrawString(VARPTR$("Rectangles"))

  _SetSolidPenPat(7)
  _SetRect(VARPTR$(MyRect%),170,10,270,
  _PaintRect(VARPTR(MyRect%))

  _SetSolidPenPat(8)
```



Overflow error in Statement.

```
DIM     Flags%(8192)
FOR Iterations% = 1 TO 10
  Count% = 0
  FOR I% = 0 TO 8190
    IF NOT Flags%(I%) THEN GOTO SkipIt
    Prime% = I% + I% + 3
    K% = I% + Prime%
    WHILE K% <= 8190
      FLAGS%(K%) = 0
      K% = K% + Prime%
    UNTIL
```

Advanced Integrated Editor:

- Integrated with the compiler to make programming fast and efficient.
- Open up to four windows at once. Each containing a different program.
- Easy to learn and use desktop editing.
- Supports Integer, Double Integer, Long Integer, Single Real and Double Real numeric types.

Compiler:

- Compiles 6,000 + lines per minute.
- Compiles to Disk or Memory utilizing all available memory.
- Incredible compilation speed retains the interactiveness of an interpreter.
- Creates stand-alone applications in seconds without leaving the TML BASIC environment.
- Complete IIGS Toolbox support.

System Requirements:

- An Apple IIGS with only 512K total memory and one 3.5" disk drive.

Language Features:

- Advanced statements like PRINT USING, IF/THEN/ELSE, DO/WHILE/UNTIL, and more.
- Allows for user-defined procedures and functions using parameters, local variables and recursion.
- Supports separate compilation of libraries.
- Supports alphanumeric labels.

Built-in Debugger:

- Built-in debugger detects and highlights runtime errors interactively.

Apple IIGS Toolbox Support:

• QuickDraw Graphics	• Sound
• Windows	• Fonts
• Dialogs	• Printing
• Menus	• Controls
• Sound	• SANE
• Event Manager	• Standard File
• Note Synthesizer	• Line Edit
• Text Tools	• and more ...

TML Systems, Inc • 8837-B Goodbys Executive Drive • Jacksonville, Florida 32217